Incremental Inference for Probabilistic Programs

Marco Cusumano-Towner
MIT, USA
marcoct@mit.edu

Benjamin Bichsel
ETH Zurich, Switzerland
benjamin.bichsel@inf.ethz.ch

Timon Gehr
ETH Zurich, Switzerland
timon.gehr@inf.ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Vikash K. Mansinghka
MIT, USA
vkm@mit.edu

Abstract

We present a novel approach for approximate sampling in probabilistic programs based on incremental inference. The key idea is to adapt the samples for a program P into samples for a program Q, thereby avoiding the expensive sampling computation for program Q. To enable incremental inference in probabilistic programming, our work: (i) introduces the concept of a trace translator which adapts samples from P into samples of Q, (ii) phrases this translation approach in the context of sequential Monte Carlo (SMC), which gives theoretical guarantees that the adapted samples converge to the distribution induced by Q, and (iii) shows how to obtain a concrete trace translator by establishing a correspondence between the random choices of the two probabilistic programs. We implemented our approach in two different probabilistic programming systems and showed that, compared to methods that sample the program Q from scratch, incremental inference can lead to orders of magnitude increase in efficiency, depending on how closely related P and Q are.

CCS Concepts • **Mathematics of computing → Markov-chain Monte Carlo methods**; **Sequential Monte Carlo methods**;

Keywords Probabilistic programming, sequential Monte Carlo, incremental computation

ACM Reference Format:
Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K. Mansinghka. 2018. Incremental Inference for Probabilistic Programs. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3192366.3192399

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192399

1 Introduction

Probabilistic models occur in many application domains such as machine learning, robotics, networking, and security [16, 25, 30, 42]. Probabilistic programming languages promise to simplify the process of working with probabilistic models by providing high-level languages for expressing these models. Because of this promise, over the last few years, we have seen an increased interest in different probabilistic languages and frameworks, including Church [18], Stan [6], R2 [35], WebPPL [19], Venture [29], Anglican [45], and PSI [17].

Key Challenge A key roadblock to the wider adoption of probabilistic programming is that general-purpose efficient inference is an intrinsically hard problem, even using approximate (typically sampling-based) methods [10, 12]. In particular, when a probabilistic model is conditioned on observations, the resulting posterior distribution may be complex and difficult to navigate or summarize.

Existing approaches address this challenge by taking advantage of analytical solutions [17, 33], possibly to subproblems [32], adaptive inference [11, 26], custom inference programs [29], scalable parallel implementation [31] or inference based on sequential observations [37, 45]. However, given the inference results for one program, inference for a second, related program may be substantially easier. Therefore, we address the following *incremental inference* task:

Given two probabilistic programs P and Q, and samples of P obtained using an existing inference algorithm, generate samples for Q by leveraging the samples for P.

Applications Incremental inference arises naturally in probabilistic modeling and inference, as one may often explore different variants of a model, change the data upon which a model is conditioned, or change the prior assumptions [40]. Incremental inference can also be used when program modifications originate from an automated process like synthesis [36] or learning [20].

This Work In this work, we propose a new approach for incremental inference in the context of probabilistic programming. The key idea is the concept of a *trace translator*, which adapts samples (traces) of P into samples (traces) of Q. To guarantee that samples obtained for Q with a trace

translator indeed represent the distribution of Q, we formalize our approach in terms of sequential Monte Carlo (SMC, [13]). We then show how to construct a trace translator by establishing a correspondence between the random choices in P and Q, and how to derive a trace translator from program edits. These steps are aided by the fact that we work with a high-level programming language where similarities between programs are easier to observe.

Main Contributions Our main contributions are:

- The concept of a trace translator which adapts samples (traces) of one program into traces of another program. By phrasing this concept in terms of classic sequential Monte Carlo theory, we ensure the adapted traces converge to the distribution induced by Q, as more traces are translated (Section 4).
- A concrete instantiation of the trace translator concept based on correspondence of random choices used in P and Q (Section 5) and a specific method to derive such correspondence from program edits (Section 6).
- An end-to-end implementation and evaluation of our approach. Our experimental results indicate that incremental inference can lead to substantial performance gains compared to sampling from scratch (Section 7).

Our approach complements standard inference: if the posterior distributions of programs P and Q are close enough, then one can use our incremental approach, otherwise, one can proceed with standard non-incremental inference.

2 Overview

In this section we informally introduce our approach on the example shown in Figure 1. A formal description, and evaluation on realistic hard inference problems are provided in later sections. In the figure, we have two versions of a classic program (see e.g. [7, 17, 22, 36]) capturing the following story. Mr. Holmes receives a telephone call by Mary, notifying him that she just woke, and is not sure if she heard a burglary alarm. Mr. Holmes wants to determine the (posterior) probability that – given that Mary woke up – there really is a burglary in progress. In the program on the left, Mr. Holmes only takes into account the possibility of a burglary, an alarm, and of Mary waking up. Here, `burglary` = $\texttt{flip}_\alpha(0.02)$ means variable `burglary` is 1 with probability 0.02 and 0 with probability 0.98. We use α as an index for the random choice made by $\texttt{flip}(0.02)$. This allows us to refer to the random choice that produced a certain value in the trace. Similarly, `alarm` = $\texttt{flip}_\beta(\texttt{pAlarm})$ means `alarm` is 1 with probability `pAlarm`, where `pAlarm` is 0.9 or 0.01, depending on whether there is a burglary or not. The line $\texttt{observe}_o(\texttt{flip}(\texttt{pMaryWakes}) == 1)$ expresses that Mary wakes up with probability `pMaryWakes`, and that we observe this event. Intuitively, observing that Mary woke up changes the probability of there being a burglary, because a burglary (indirectly) makes it more likely that Mary wakes up. In the (refined) program on the right, Mr. Holmes refines the model to also take into account the possibility the alarm was triggered by an earthquake.

Motivation for Trace Translation To estimate the posterior probability of a burglary in the refined program, we can generate traces from the posterior of that program, counting how often we see a burglary. This amounts to performing inference in the refined program. The box plots in Figure 1 show that in both programs, a burglary is roughly 10 times more likely to occur in the posterior than in the prior, hence simple rejection sampling using the prior as a proposal will be inefficient. In contrast, the posterior distributions represented by both programs are very similar (in the refined program, the probability of there being a burglary slightly decreases). This suggests that if we already have sampled many traces from the posterior of the original program, it may be cheaper to simply adapt them to the new program. That is, to *translate* existing traces of the original program into traces of the modified program.

Trace Translation Concretely, the bottom half of Figure 1 shows the translation process for one of the traces of the original program $t = [\alpha \mapsto 1, \beta \mapsto 1]$. Here, $\alpha \mapsto 1$ indicates that the program sampled value 1 for $\texttt{flip}_\alpha(0.02)$. For convenience, we also show the observation o in our visualization of the trace, even though it is not formally part of the trace. Next to the trace, we show the probabilities for each random choice. For example, $p_\alpha = 0.02$ indicates that $\texttt{flip}_\alpha(0.02)$ evaluates to 1 with probability 0.02. Before translation starts, we establish a correspondence f between random choices in the two programs. In this example, α corresponds to α', indicating that $\texttt{flip}_\alpha(0.02)$ and $\texttt{flip}_{\alpha'}(0.02)$ play a similar role in both programs.

Our trace translator then adapts the trace t into trace $u = [\alpha' \mapsto 1, \gamma' \mapsto 1, \beta' \mapsto 1]$. Because α and α' are in correspondence, the trace translator reuses the random choice made by α as the random choice for α', and likewise for β and β'. For γ', there is no corresponding random choice in the original trace. Hence, the trace translator selects the value of γ' by sampling $\texttt{flip}_{\gamma'}(0.005)$.

Weighting Traces Generally, the distribution induced by our trace translator may not exactly match the actual distribution induced by the modified program. To compensate for this difference, we introduce *weights* for our traces. Intuitively, traces sampled too frequently are assigned a low weight, while traces sampled too infrequently are assigned a higher weight. When we average over traces (for example to estimate the probability of an event), traces with higher weight have higher contribution. In our example, trace u is slightly more likely to be generated by the modified program than by our trace translator, and hence gets weighted by $w' \approx 1.19$. Intuitively, this is because the random choice $\beta \mapsto 1$ is less likely than the random choice $\beta' \mapsto 1$, but

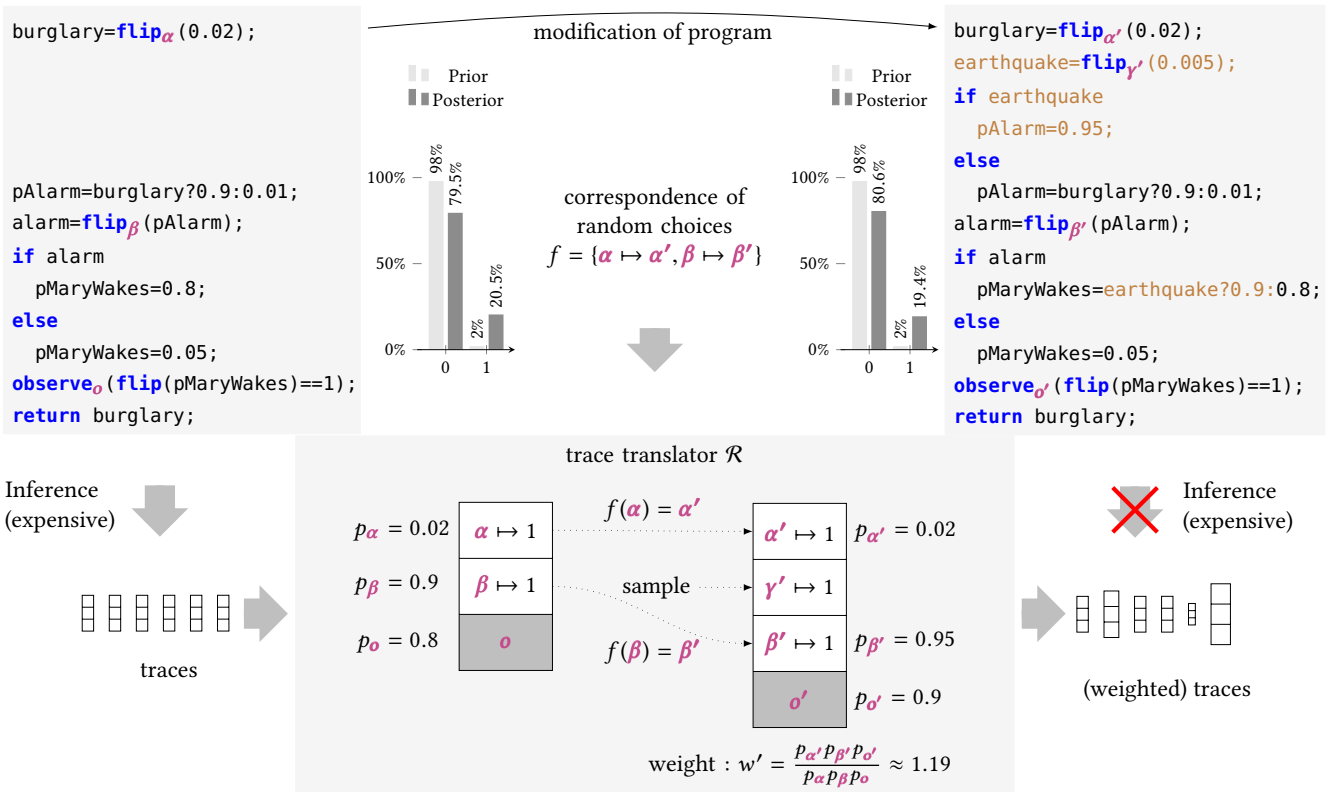

Figure 1. Trace translation from an original program to a modified program. In both programs, we have indexed random choices by letters $\alpha, \beta, \alpha', \beta', \gamma'$ and observations by letters o, o'. The bar graphs show the prior distribution (on `burglary`) and posterior distribution (after observing that Mary wakes up with probability `pMaryWakes`) of the original (left) and the modified (right) program. The trace translator adapts approximate posterior samples (i.e. traces) for the program on the left into approximate posterior samples for the program on the right. The size of the traces on the right represents their weight.

our translator always takes the value for β as the value for β'. Similarly, the observation that Mary wakes up is slightly more likely in the modified program ($p_{o'} = 0.9$) than in the original program ($p_o = 0.8$), meaning that u should get a higher weight.

Using weights to compensate for the difference between an actual distribution and a desired distribution is the basis of many Monte Carlo methods including importance sampling, and we adapt this approach to the probabilistic programming setting. Taking into account the weights allows us to aggregate the translated traces to compute properties of the modified program, such as the probability of some event occurring (see Lemma 2, Section 4.2). If unweighted traces from the modified program are desired, one can convert weighted traces to unweighted traces by *resampling* (see Section 4.2).

Discussion At a first sight, it may seem as if sampling from the modified program can be achieved in a simpler way, by leveraging existing techniques and without making use of a trace translator. An alternative approach could be to use the traces from the original program as a starting point for samplers such as Markov chain Monte Carlo (MCMC). Unfortunately, this is not possible for programs such as the one in Figure 1, as traces from the original program are inconsistent with the modified program (they are missing the random choice γ'). Further, such an approach would be less efficient as it ignores knowledge encoded in our weight.

Overall, incremental inference with trace translation can be seen as a valuable optimization over standard sampling: if the distributions of the two programs are believed to be similar, one can use translation to compute the posterior distribution of the target program, otherwise, one can employ standard samplers as usual to estimate that distribution.

Paper Structure We first introduce a probabilistic programming language with a semantics based on traces (Section 3). Then, we introduce the concept of a *trace translator* and an incremental SMC inference algorithm based on this concept (Section 4). We introduce a concrete trace translator that makes use of a correspondence between the original and the modified program (Section 5) and show how this correspondence can be derived from program edits (Section 6). Finally, we provide a detailed evaluation of incremental inference based on trace translation (Section 7).

$$\frac{}{(P[x],\sigma) \xrightarrow[1]{[]} (P[\sigma(x)],\sigma)} \qquad \frac{}{(P[\ominus v],\sigma) \xrightarrow[1]{[]} (P[\text{eval}(\ominus v)],\sigma)} \qquad \frac{}{(P[\texttt{flip}(v)],\sigma) \xrightarrow[v]{[1]} (P[1],\sigma)}\ v \in [0,1]$$

$$\frac{}{(x = v,\sigma) \xrightarrow[1]{[]} (\texttt{skip},\sigma[x \mapsto v])} \qquad \frac{(P_1,\sigma) \xrightarrow[p]{t} (P_1',\sigma')}{(P_1;P_2,\sigma) \xrightarrow[p]{t} (P_1';P_2,\sigma')} \qquad \frac{}{(\texttt{skip};P_2,\sigma) \xrightarrow[1]{[]} (P_2,\sigma)}$$

$$\frac{}{(\texttt{if}\ v\ \{P_1\}\ \texttt{else}\ \{P_2\},\sigma) \xrightarrow[1]{[]} (P_1,\sigma)}\ v \neq 0 \qquad \frac{}{(\texttt{observe}(\texttt{flip}(v) == 1),\sigma) \xrightarrow[v]{[]} (\texttt{skip},\sigma)}$$

Figure 2. Semantics for expressions and programs. For the rule for $P[\ominus v]$, $\text{eval}(\ominus v)$ evaluates the unary operation $\ominus$ on v. We do not mention all rules, but the remaining rules are straight-forward. For example, the rule to evaluate $P[v_1 \oplus v_2]$ is analogous to the rule to evaluate $P[\ominus v]$. $\sigma[x \mapsto v]$ updates σ to v at x, and $\sigma(x)$ evaluates σ at x.

3 A Probabilistic Language

We now introduce a simple probabilistic language and its semantics. The language will be used to formalize the notion of incremental inference for probabilistic programs. To ease presentation, we omit functions, continuous distributions and loops. These can be included if needed (we discuss continuous distributions at the end of this section).

Syntax The syntax of our programming language describes expressions (E), random expressions (R) and programs (P):

$$\begin{aligned} E &::= v \mid x \mid \ominus E \mid E_1 \oplus E_2 \mid R \\ R &::= \texttt{flip}(E) \mid \texttt{uniform}(E_1, E_2) \\ P &::= \texttt{skip} \mid x = E \mid P_1; P_2 \mid \texttt{observe}(R == E) \mid \\ &\quad\ \texttt{if}\ E\ \{P_1\}\ \texttt{else}\ \{P_2\} \end{aligned}$$

Expressions (E) include a rational constant (v), a variable (x), unary and binary operators ($\ominus$, $\oplus$) and random expression (R). The latter include random choices resulting in 1 or 0 with probability E and $1 - E$ respectively (via $\texttt{flip}(E)$), and selecting an integer between E_1 and E_2 uniformly at random (via $\texttt{uniform}(E_1, E_2)$). Programs include standard deterministic constructs, and the statement $\texttt{observe}(R == E)$, which decreases the probability of specific traces. For example, $\texttt{observe}(\texttt{flip}(p) == x)$ decreases the probability of the current trace by a factor of p if x evaluates to 1, and by a factor of $1 - p$ if x evaluates to 0. We refer to statements of the form $\texttt{observe}(R == E)$ as *observations*. Note that only the outcome of random expressions R can be observed (for example, we cannot write $\texttt{observe}(x = 1)$). This restriction is commonly used in sampling-based probabilistic programming languages (e.g. [29]). We express Boolean values in terms of rational numbers: 0 stands for `false`, while all other values stand for `true`.

Small-step Semantics We provide small-step operational semantics for our language. We use $P[\square]$ to refer to a program that contains a hole. The hole is always at the location of the next expression to be evaluated, making sure the evaluation order is clear. For example, $P[\square]$ defined by $i = i + \square$ is invalid because for $E_1 + E_2$, we evaluate E_1 before E_2. $P[E]$ stands for P where the hole is replaced by expression E. For example, for $P[\square]$ defined by "$i = \square + 1$", $P[i]$ is "$i = i + 1$".

For a program P and a state σ, $(P,\sigma) \xrightarrow[p]{t} (P',\sigma')$ means that one step of the evaluation of P in state σ produces trace t with probability $p \in [0,1]$. After this particular step, the remaining program to be executed is P' in state σ'. Here, σ is a map from variables to values, e.g. $\sigma = \{x \mapsto 3, y \mapsto \frac{1}{2}\}$.

Figure 2 shows the semantic rules. Note that for many rules, we assume that sub-expressions are just a value $v \in \mathbb{Q}$. This is because the small-step semantics will first evaluate that sub-expression to v and only then apply the rule. We intentionally do not include a rule that handles the program `skip`, thus, `skip` marks the end of a program execution.

Trace Definition A *trace* of a program is a collection of values taken from every random expression evaluated during program execution, that is, $t \in \mathcal{T} := \bigcup_{n \in \mathbb{N}} \mathbb{Q}^n$. The classic notion of a trace (containing the state of the program at each point in time) can be reconstructed from our notion of a trace. Likewise, the return value and the result of all observations can be reconstructed from a trace. Because we think of Q as inducing a probability distribution, we often refer to traces of Q as *samples* of Q.

Note that every element (random choice) of the trace corresponds to a random expression R in the program. We denote the indices of random choices in a trace t by the set R_t. Because our language does not support loops, there is at most one random choice for every random expression in a program. For other languages that do support loops, random choices can by indexed by a specific naming scheme, see for example [44]. For $i \in R_t$, we denote the result of random choice i in t by t_i. Likewise, O_t is the set of indices of observations evaluated for t. Note that O_t can be reconstructed from the trace. Additionally, for $i \in R_t$ or $i \in O_t$, $t_{1:i-1}$ are the results of all random choices before i.

If a variable is only assigned to once in every execution of the program, we can index entries in the trace by variable names. For example, $x = \texttt{flip}(1/2); y = \texttt{flip}(1/2);$ can be

written as $t = [x \mapsto 1, y \mapsto 0]$. In this example, $R_t = \{x, y\}$, and $t_{1:y-1} = [x \mapsto 1]$.

Probability of a Trace To run a program P_0 from state σ_0, we apply small-step transitions. For a sequence of small-step transitions $(P_0, \sigma_0) \xrightarrow[p_0]{t_0} (P_1, \sigma_1) \xrightarrow[p_1]{t_1} \cdots \xrightarrow[p_n]{t_n} (\texttt{skip}, \sigma_n)$, we write:

$$(P_0, \sigma_0) \xRightarrow[p_0 \cdot p_1 \cdots p_n]{t_0 +\!\!+ t_1 +\!\!+ \cdots +\!\!+ t_n} \sigma_n$$

Here, $+\!\!+$ denotes trace concatenation and the above equation means that we ran P_0 from state σ_0 and produced a trace $t = t_0 +\!\!+ t_1 +\!\!+ \cdots +\!\!+ t_n$ with (sub-)probability $p = p_0 \cdot p_1 \cdots p_n$, ending in state σ_n. In this case, we define the *unnormalized probability of trace t*, denoted as $\widetilde{\Pr}[t \sim P]$, by p. To make sure that $\widetilde{\Pr}[t \sim P]$ does not depend on σ_0, we assume that our programs initialize all variables before first use.

Let $\Pr[t_i \sim P \mid t_{1:i-1}]$ be the probability that random expression i chooses value t_i given the evaluation of all random expressions before i. Similarly, let $\Pr[i \sim P \mid t_{1:i-1}]$ be the probability of satisfying observation i given evaluations $t_{1:i-1}$. Then,

$$\widetilde{\Pr}[t \sim P] = \prod_{i \in R_t} \Pr[t_i \sim P \mid t_{1:i-1}] \cdot \prod_{i \in O_t} \Pr[i \sim P \mid t_{1:i-1}]$$

Let $\mathcal{T}_P$ denote the set of all traces for program P. The *normalizing constant* for P is Z_P defined by $\sum_{t \in \mathcal{T}_P} \widetilde{\Pr}[t \sim P]$. In the absence of observations, $Z_P = 1$.

$\Pr[t \sim P] := \frac{\widetilde{\Pr}[t \sim P]}{Z_P}$ denotes the probability distribution that results after normalization of $\widetilde{\Pr}[t \sim P]$. If t is sampled according to this distribution, we write $t \sim P$.

Example 1. Consider the program P in Figure 3. A possible trace of P is $t = [1, 4, 1]$. For readability, we can index t by variable names, resulting in $[b \mapsto 1, c \mapsto 4, d \mapsto 1]$. The unnormalized probability of t is given by $\widetilde{\Pr}[t \sim P] = \frac{1}{3} \cdot \frac{1}{6} \cdot \frac{1}{2} \cdot \frac{1}{5}$, where we have a factor for the random choice of b, c and d, and a factor for the observation. Summing over all traces of P yields $Z_P = 0.7$ (i.e., the probability of satisfying the observation is 0.7). Hence, the normalized probability of t is $\Pr[t \sim P] = \frac{\widetilde{\Pr}[t \sim P]}{0.7}$.

```
a = 1;
b = flip(a/3);
if a<2
  c = uniform(1,6);
else
  c = uniform(6,10);
d = flip(b/2);
observe(flip(1/5)==d);
```

Figure 3. Simple probabilistic program.

Continuous Distributions In order to handle a purely continuous setting (without discrete random choices), the necessary changes to our semantics are minimal. First, we capture the small-step semantics by $(P, \sigma) \xrightarrow[f]{t} (P', \sigma')$, meaning one step of the evaluation of P in state σ produces trace t with density $f \in \mathbb{R}_{\geq 0}$ (by using densities, we avoid issues with zero-probability traces). Second, we compute the normalizing constant by integration over all possible traces (instead of summation). Note that since different traces can be of different length, we have to sum over all possible trace lengths. Adapting the trace translator to the purely continuous setting requires interpreting the forward kernel (introduced in Section 4) $k_{P \to Q}(u; t)$ as a family of densities on traces u, and likewise for the backward kernel.

A formal semantics for programs with *both* continuous and discrete random choices requires measure theory. To avoid complicating this work substantially with the increased complexity of measure theory, we informally treat continuous and discrete random choices the same (leading to multiplication of probabilities and densities). This is a common approach in sampling-based probabilistic programming systems, see e.g. [29].

4 SMC for Probabilistic Programs

This section introduces a general approach that uses the results of sampling-based inference for a given probabilistic program P in order to make inference for a new probabilistic program Q more efficient. The main idea is to transform the set of traces generated by the inference algorithm for P into a set of traces of Q in a way where these traces are accurate approximate samples of Q.

To ensure the transformed samples follow the distribution induced by Q arbitrarily closely, we phrase this transformation in terms of sequential Monte Carlo (SMC, [13]) theory. To apply this theory in the context of probabilistic programs, we introduce the concept of a trace translator.

4.1 Trace Translators

If we can efficiently transform posterior samples for one inference problem into posterior samples for a second inference problem, we achieve efficient incremental inference. We formalize this notion of a transformation using the concept of a *trace translator*. Formally, a trace translator $\mathcal{R}$ is a tuple:

$$\mathcal{R} = (P, Q, k_{P \to Q}, \ell_{Q \to P})$$

where P and Q are probabilistic programs expressed in the language of Section 3. The *forward kernel* $k_{P \to Q}(u; t)$ of the translator defines a distribution on traces u of program Q for each possible trace t of program P. Intuitively, $k_{P \to Q}$ translates traces of P to traces of Q. In our illustration of trace translators (Figure 1), $k_{P \to Q}(u; t) = 0.005$, because $k_{P \to Q}$ translates the trace $t = [\alpha \mapsto 1, \beta \mapsto 1]$ to the trace $u = [\alpha' \mapsto 1, \gamma' \mapsto 1, \beta' \mapsto 1]$ by reusing the random choices made by α and β and sampling from the random choice γ' (sampling 1 with probability 0.005). The *output distribution* $\eta_{P \to Q}(u)$ of the translator is the probability that a trace translator produces u, if we assume that its input

trace t was itself sampled from the program P:

$$\eta_{P\to Q}(u) := \sum_{t\in\mathcal{T}_P} \Pr[t \sim P]\; k_{P\to Q}(u;t)$$

In Figure 1, $\eta_{P\to Q}(u) = \frac{1}{Z_P}\cdot 0.02\cdot 0.9\cdot 0.8\cdot 0.005$ (the sum disappears because $k_{P\to Q}(u;t') = 0$ for all $t' \neq t$). Here, $\frac{1}{Z_P}$ comes from normalization, 0.02 and 0.9 come from $\texttt{flip}_\alpha(0.02)$ and $\texttt{flip}_\beta(0.9)$, 0.8 comes from the observation $\texttt{observe}_o(\texttt{flip}(\texttt{pMaryWakes}) == 1)$, and $0.005 = k_{P\to Q}(u;t)$.

A *perfect trace translator* has the posterior for program Q as its output distribution, that is, $\eta_{P\to Q}(u) = \Pr[u \sim Q]$. In general, perfect trace translators are not feasible, and we resort to approximate trace translators. For approximate trace translators, the *weight* is a function on traces u of Q that corrects for the difference between $\eta_{P\to Q}(u)$ and $\Pr[u \sim Q]$, by assigning larger weight to traces u that are generated less often under $\eta_{P\to Q}$ than under the posterior $\Pr[u \sim Q]$:

$$w_{P\to Q}(u) := \frac{\Pr[u \sim Q]}{\eta_{P\to Q}(u)} = \frac{\Pr[u \sim Q]}{\sum_{t\in\mathcal{T}_P}\Pr[t \sim P]\; k_{P\to Q}(u;t)} \quad (1)$$

In Figure 1, $w_{P\to Q}(u) = \frac{1/Z_Q\cdot 0.02\cdot 0.005\cdot 0.95\cdot 0.9}{1/Z_P\cdot 0.02\cdot 0.9\cdot 0.8\cdot 0.005} \approx \frac{Z_P}{Z_Q}\cdot 1.19$.

The sum in the denominator of Equation (1) is often intractable to compute. In this case, the weight can be replaced by a *weight estimate*, which is the following function on traces u and t:

$$\hat{w}_{P\to Q}(u;t) := \frac{\widetilde{\Pr}[u \sim Q]\; \ell_{Q\to P}(t;u)}{\widetilde{\Pr}[t \sim P]\; k_{P\to Q}(u;t)} \quad (2)$$

where $\ell_{Q\to P}$ is a *backward kernel* that defines a distribution on traces t of program P, for each possible trace u of program Q. Given u, the weight estimate is an unbiased estimate of $\frac{Z_Q}{Z_P} w_{P\to Q}(u)$ (see Appendix A of the supplementary material). Hence, the weight estimate is in expectation proportional to the weight (proportionality is enough to ensure the theoretical guarantees of SMC). The additional factor $\frac{Z_Q}{Z_P}$ is due to the unnormalized probabilities $\widetilde{\Pr}[u \sim Q]$ and $\widetilde{\Pr}[t \sim P]$ in Equation (2), which are substantially easier to compute than their normalized versions. In Figure 1, $\ell_{Q\to P}(t';[\alpha' \mapsto a, \gamma' \mapsto c, \beta' \mapsto b])$ is a point distribution that is 1 if and only if $t' = [\alpha \mapsto a, \beta \mapsto b]$. This choice of $\ell_{Q\to P}$ yields $\hat{w}_{P\to Q}(u;t) \approx 1.19$.

The optimal backward kernel is the one that reduces $\hat{w}_{P\to Q}(u;t)$ to $\frac{Z_Q}{Z_P} w_{P\to Q}(u)$:

$$\ell^{OPT}_{Q\to P}(t;u) := \frac{\Pr[t \sim P]\; k_{P\to Q}(u;t)}{\eta_{P\to Q}(u)} \quad (3)$$

The procedure TRANSLATE in Algorithm 1 defines the operation of a trace translator.

Trace Translator Error The error of an approximate trace translator for programs P and Q depends on both $k_{P\to Q}$ and

Algorithm 1 Abstract Trace Translator

Require: Trace translator $\mathcal{R} = (P, Q, k_{P\to Q}, \ell_{Q\to P})$,
Input trace t of program P

procedure TRANSLATE($\mathcal{R}$, t)
  $u \sim k_{P\to Q}(\cdot;t)$ ▷ Generate trace for program Q
  $w \leftarrow \hat{w}_{P\to Q}(u;t)$ ▷ Evaluate weight
  return (u, w)
end procedure

$\ell_{Q\to P}$, and is quantified using Kullback-Leibler (KL) divergence. Recall that the KL divergence between two distributions μ and ν on a discrete set $\mathcal{X}$ is defined as:

$$D_{KL}(\mu \,\|\, \nu) := \sum_{x\in\mathcal{X}} \mu(x) \log\left(\frac{\mu(x)}{\nu(x)}\right)$$

The KL divergence is zero if and only if the distributions are the same. We define the *trace translator error* $\epsilon(\mathcal{R})$ as:

$$\epsilon(\mathcal{R}) := D_{KL}\left(Q \,\middle\|\, \eta_{P\to Q}\right) + \mathbb{E}_{u\sim Q}\left[D_{KL}\left(\ell_{Q\to P}(\cdot;u) \,\middle\|\, \ell^{OPT}_{Q\to P}(\cdot;u)\right)\right] \quad (4)$$

The first term in Equation (4) accounts for the difference between the output distribution $\eta_{P\to Q}$ and the posterior for program Q, and the second term accounts for any error in our estimate of the weight.

4.2 SMC with Trace Translators

The trace translator primitive can be used to construct incremental inference algorithms. For programs P and Q, INFER in Algorithm 2 gives an incremental inference algorithm based on SMC that takes as input a weighted collection $\{(t_j, w_j)\}_{j=1}^M$ of traces of P that approximates the posterior distribution $\Pr[t \sim P]$, and returns a new weighted collection $\{(u'_j, w'_j)\}_{j=1}^M$ of traces of Q that approximates the new posterior distribution $\Pr[u \sim Q]$. Concretely, a weighted collection of traces $\{(t_j, w_j)\}_{j=1}^M$ approximates $\Pr[t \sim P]$ by a distribution $\hat{P}(t)$ of the form:

$$\hat{P}(t) := \sum_{j=1}^{M} \frac{w_j}{\sum_{k=1}^M w_k}\delta(t, t_j) \approx \Pr[t \sim P]$$

where δ is the Kronecker delta. The output approximation $\hat{Q}(u) \approx \Pr[u \sim Q]$ is defined analogously in terms of weighted traces $\{(u'_j, w'_j)\}_{j=1}^M$. Algorithm 2 works by first translating all M traces of P into traces of program Q using TRANSLATE, and then updates the weights of the translated traces. Next, it optionally resamples the translated traces and sets their weights to 1. Finally, it runs $mcmc_Q$, an MCMC (Markov Chain Monte Carlo) sampler for Q, to produce a new trace u'_j from trace u_j with probability $mcmc_Q(u'_j; u_j)$ (by running an independent Markov chain for each j). We illustrate Algorithm 2 in Figure 4. Algorithm 2 can be interpreted as a single step of a sequential Monte Carlo sampler [13], applied to generic sequences of probabilistic programs.

Algorithm 2 A single step of SMC for probabilistic programs

Require: Trace translator $\mathcal{R} = (P, Q, k_{P\to Q}, \ell_{Q\to P})$,
MCMC sampler $mcmc_Q$ for program Q
Weighted traces $\{(t_j, w_j)\}_{t=1}^{M}$ of program P

procedure INFER($\mathcal{R}$, $mcmc_Q$, $\{(t_j, w_j)\}_{j=1}^{M}$)
  for $j \leftarrow 1 \ldots M$ **do**
    $(u_j, \Delta w_j) \sim$ TRANSLATE($\mathcal{R}, t_j$)
    $w'_j \leftarrow w_j \cdot \Delta w_j$ ▷ Update weight
  end for
  $\{(u_j, w'_j)\}_{j=1}^{M} \leftarrow$ RESAMPLE($\{(u_j, w'_j)\}_{j=1}^{M}$) ▷ (Optional)
  for $j \leftarrow 1 \ldots M$ **do**
    $u'_j \leftarrow mcmc_Q(\cdot; u_j)$ ▷ MCMC for Q
  end for
  return $\{(u'_j, w'_j)\}_{j=1}^{M}$
end procedure
procedure RESAMPLE($\{(u_j, w_j)\}_{j=1}^{M}$)
  for $j \leftarrow 1 \ldots M$ **do**
    $a_j \sim$ Categorical($[w_1, \ldots, w_M] / \sum_{k=1}^{M} w_k$)
    $u'_j \leftarrow u_{a_j}$
  end for
  return $\{(u'_j, 1)\}_{j=1}^{M}$
end procedure

***Multiple Steps and* RESAMPLE** Often, programs are modified in an iterative process, and thus, it may be desirable to generate traces from a *sequence of programs*. In this case, we can run Algorithm 2 repeatedly, once for each new program in the sequence, to iteratively transform the weighted collection of traces from one program to the next. However, after several iterations, only a small fraction of traces in the collection will likely have appreciable weight, reducing the effective number of traces.[1] To remedy this, we can optionally execute RESAMPLE, which picks M traces (with replacement), with a probability proportional to their weight, producing a collection of new traces that each have weight 1. Formally, when sampling a_j from Categorical($[w_1, \ldots, w_M] / \sum_{k=1}^{M} w_k$), a_j will be set to x with probability $w_x / \sum_{k=1}^{M} w_k$. This means that traces with a high weight (i.e. more representative traces) will get chosen more often than traces with low weight. By duplicating high weight traces, we re-allocate our computational resources to those traces that are more representative of the posterior so that future steps will be more efficient. One can also use the weights to detect when an incremental approach may not be feasible, by monitoring the 'effective sample size' [28].

Interoperability with MCMC Sampling Using MCMC after trace translation in INFER can increase the quality of the output approximation, but is not necessary to achieve formal guarantees (see Lemma 2). The algorithm is amenable

[1] This is called particle degeneracy in the SMC literature. Other resampling schemes besides independent resampling are also possible.

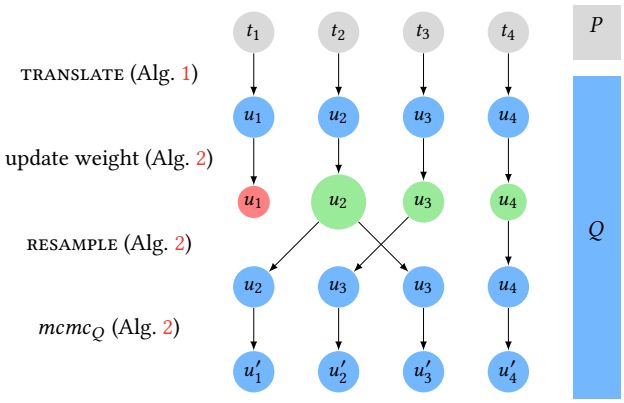

Figure 4. Illustration of Algorithm 2 for $M = 4$. Each circle stands for a trace. The size of the circle represents its weight. Traces $t_{1:4}$ belong to program P and have equal weight, while traces $u_{1:4}$ and $u'_{1:4}$ belong to program Q. RESAMPLE removes u_1, and duplicates u_2.

to a modular design in which the trace translation and the MCMC samplers are implemented independently. To use an MCMC sampler $mcmc_Q$ with Algorithm 2, we impose the (standard) requirement that it must admit the posterior $\Pr[u \sim Q]$ as an invariant distribution:

$$\sum_{u \in \mathcal{T}_Q} \Pr[u \sim Q] \cdot mcmc_Q(u'; u) = \Pr[u' \sim Q] \quad \text{for all} \quad u' \in \mathcal{T}_Q$$

This allows interoperation with various MCMC samplers, including generic single site Metropolis-Hastings [18, 44], Gibbs sampling, and custom user-specified MCMC algorithms [29]. Note that one call to $mcmc_Q$ can lead to multiple iterations of an MCMC sampler: the input u is the initial state of the Markov chain and u' is the state after the last iteration.

***Using the Output of* INFER** The weighted collection of traces $\{(u'_j, w'_j)\}_{j=1}^{M}$ returned by Algorithm 2 can be used to estimate the expectation $\mathbb{E}_{u\sim Q}[\varphi(u)]$ of any property (captured by a function) $\varphi\colon \mathcal{T}_Q \to \mathbb{R}$ under the posterior distribution for program Q, using:

$$\frac{\sum_{j=1}^{M} \varphi(u'_j) w'_j}{\sum_{j=1}^{M} w'_j} \approx \mathbb{E}_{u\sim Q}[\varphi(u)] \tag{5}$$

For example, to estimate the probability of an event $A \subseteq \mathcal{T}_Q$, we use $\varphi(u) := \mathbf{1}[u \in A]$ (the indicator function of set A) , which gives the estimate:

$$\frac{\sum_{j=1}^{M} w'_j \mathbf{1}[u'_j \in A]}{\sum_{j=1}^{M} w'_j} \approx \sum_{u \in A} \Pr[u \sim Q]$$

If unweighted approximate posterior samples from Q are desired, the weighted collection $\{(u'_j, w'_j)\}_{j=1}^{M}$ can be converted to an unweighted collection using RESAMPLE.

Formal Guarantees The following lemma states that by translating sufficiently many traces, the estimator in Equation (5) produced using weighted collections of traces from INFER will converge to the true expectation.

Lemma 2. *Let $\mathcal{R} = (P, Q, k_{P\to Q}, \ell_{Q\to P})$ be a trace translator with finite error $\epsilon(\mathcal{R})$. Let $\varphi : \mathcal{T}_Q \to \mathbb{R}$ be a function. Let $t_j \sim P$ and $w_j = 1$ for $j \in \{1, \ldots, M\}$. Let MCMC sampler $mcmc_Q$ have invariant distribution Q. Let $\{(u'_j, w'_j)\}_{j=1}^M$ be the output of* INFER *on input $(\mathcal{R}, mcmc_Q, \{(t_j, w_j)\}_{j=1}^M)$. Then, almost surely,*

$$\frac{\sum_{j=1}^M w'_j \varphi(u'_j)}{\sum_{j=1}^M w'_j} \xrightarrow[M\to\infty]{} \mathbb{E}_{u\sim Q}[\varphi(u)]$$

Note that Lemma 2 only gives guarantees in the limit of infinitely many traces. However, the number of translated traces needed to achieve a given error crucially depends on the trace translator error $\epsilon(\mathcal{R})$. If $\epsilon(\mathcal{R})$ is large, then more traces are needed; thus $\epsilon(\mathcal{R})$ determines the efficiency of Algorithm 2. Indeed, the number of traces M needed to achieve a given error in estimates of the form Equation (5) scales approximately exponentially in the trace translator error $\epsilon(\mathcal{R})$. See Appendix B of the supplementary material for more details.

When Algorithm 2 is iterated across a sequence of programs, we retain the theoretical guarantee of Lemma 2, even if we do not run MCMC up to convergence (i.e. even though the collection of weighted traces passed from one step to the next are not independent and identically distributed exact samples from the previous step) [21].

5 Trace Translator with Correspondence

The design space of trace translators for two probabilistic programs P and Q is massive and hence, designing an efficient translator relies on knowledge of how P and Q relate. This section describes the design of such a translator, one that is appropriate for cases when we can establish semantic correspondence between some of the random expressions in P and Q. Intuitively, we say two random expressions are in correspondence if we know (or believe) that they play the same or similar role in the two programs (this will be formalized below).

5.1 Forward Kernel

A (semantic) *correspondence* between two programs P and Q is a bijection $f : F_Q \to F_P$. Here, $F_Q \subseteq \cup_{t\in\mathcal{T}_Q} R_t$ is a collection of indices of possible random choices in Q, and likewise for F_P. We describe the forward kernel by giving a procedure that samples from it. That is, for a correspondence f between P and Q, and for a trace t of program P, we define $k_{P\to Q}(u; t)$ as the probability that the following procedure produces the trace u as output: Initialize an empty trace u (i.e. a trace that has no random choices in it). Execute Q, but when Q makes a random choice i for which a correspondence is given (i.e. $i \in F_Q$), look up the value for the

```
1  a=flip_α(1/2);
2  if a==0
3    b=uniform_β(0,5);
4  else
5    b=flip_γ(1/2);
6  c=flip_δ(1/2);
7
```

```
1  a=flip_ϵ(1/3);
2  if a==0
3    b=uniform_ζ(0,5);
4  else
5    b=flip_η(1/2);
6  c=uniform_θ(1,6);
7  d=uniform_ι(-5,-2);
```

Figure 5. Two simple programs P and Q. Random expressions are indexed by letters $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta$ and ι.

corresponding choice $f(i)$ in the trace t, and populate the trace u with this value ($u_i \leftarrow t_{f(i)}$). If $i \notin F_Q$, sample u_i by evaluating the appropriate random expression in Q. Finally, return u. If u agrees with t on the corresponding random choices then $k_{P\to Q}(u; t)$ is the product of the probabilities of each non-corresponding random choice. If u does not agree with t then $k_{P\to Q}(u; t) = 0$:

$$k_{P\to Q}(u; t) = \prod_{i\in R_u \setminus F_Q} \Pr[u_i \sim Q \mid u_{1:i-1}] \prod_{i\in R_u \cap F_Q} \delta(u_i, t_{f(i)}) \tag{6}$$

where δ is the Kronecker delta.

There are two cases when using the correspondence f is not possible: (i) $f(i)$ is not present in t where $i \in F_Q$ and i is a random choice occurring in u (this can happen because of branching), and (ii) the support of a random choice $i \in F_Q$ in u is different from the support of $f(i)$ in t. Recall that the support of a distribution is the set of all elements that have positive probability of occurring. For instance, if `uniform`$(0, x)$ is a statement in the program where during execution of a trace t, x takes the value 9, then the support of that random choice in t is $\{0, \ldots, 9\}$. We can handle both cases by dynamically detecting them and evaluating the appropriate random expression in Q, just as if no correspondence for i was available. For simplicity, we restrict our attention to the case where using the correspondence is always possible.

Example 3. Figure 5 shows two short programs P and Q, where we indexed the random choices to aid presentation. For these programs, it is reasonable to establish correspondence between the random choices as follows: $\epsilon \leftrightarrow \alpha, \zeta \leftrightarrow \beta$ and $\eta \leftrightarrow \gamma$. Note that ι is not matched because there is no corresponding random expression in P. Also, we cannot match δ and θ, even though both are assigned to the same variable `c`. This is because δ and θ have different support and matching them would mean the translator could never produce traces where, e.g, $[\theta \mapsto 6]$. We cannot match $\beta \leftrightarrow \eta$ for the same reason.

Assuming the correspondence above, consider a specific trace $t = [\alpha \mapsto 1, \gamma \mapsto 1, \delta \mapsto 1]$ of P. In the construction of the translated trace u from t, $k_{P\to Q}$ will reuse the random choices α and γ, but not δ. A possible result of the translation

is thus $u = [\epsilon \mapsto 1, \eta \mapsto 1, \theta \mapsto 3, \iota \mapsto -3]$. Formally, $k_{P\to Q}(u;t) = \frac{1}{6} \cdot \frac{1}{4}$, because during construction of u, $k_{P\to Q}$ samples from the two random expressions $\texttt{uniform}_\theta(1,6)$ and $\texttt{uniform}_\iota(-5,-2)$.

5.2 Estimating the Weight

For a trace u of Q, the set $\{t \in \mathcal{T}_P \mid k_{P\to Q}(u;t) > 0\}$ is referred to as the *explanations* of u. To evaluate the weight $w_{P\to Q}(u)$ of a trace u produced by our algorithm, we would need to sum over all possible explanations of u, as shown in Equation (1). Although there are fewer explanations than there are traces of P, the sum over all explanations of u may still be computationally intractable. To avoid computing this intractable sum, we instead compute an estimate of the weight. In particular, we use the backward kernel that translates traces from Q to P in the same way as the forward kernel which starts from Q and translates to P:

$$\ell_{Q\to P}(t;u) := k_{Q\to P}(t;u) \tag{7}$$

This choice of backward kernel means that the weight estimate $\hat{w}_{P\to Q}(u;t)$ only depends on random choices in correspondence and on observations. Concretely, using Equations (6) and (7) to replace $k_{P\to Q}(u;t)$ and $\ell_{Q\to P}(t;u)$ in $\hat{w}_{P\to Q}(u;t)$ yields the following weight estimate:

$$\frac{\prod_{i\in F_Q\cap R_u} \Pr\left[u_i \sim Q \mid u_{1:i-1}\right] \prod_{i\in O_u} \Pr\left[i \sim Q \mid u_{1:i-1}\right]}{\prod_{i\in F_P\cap R_t} \Pr\left[t_i \sim P \mid t_{1:i-1}\right] \prod_{i\in O_t} \Pr\left[i \sim P \mid t_{1:i-1}\right]} \tag{8}$$

Applying Equation (8) for the programs in Figure 5 and original trace $t = [\alpha \mapsto 1, \gamma \mapsto 1, \delta \mapsto 1]$ and translated trace $u = [\epsilon \mapsto 1, \eta \mapsto 1, \theta \mapsto 3, \iota \mapsto -3]$, the numerator contains factors for random choices ϵ and η, while the denominator contains factors for random choices α and γ. The weight estimate is:

$$\hat{w}_{P\to Q}(u;t) = \frac{\Pr[\texttt{flip}(1/3) = 1] \cdot \Pr[\texttt{flip}(1/2) = 1]}{\Pr[\texttt{flip}(1/2) = 1] \cdot \Pr[\texttt{flip}(1/2) = 1]} = 2/3$$

Note that this example does not contain any observations.

To compute the numerator of the weight estimate in (8), it suffices to execute program Q, drawing random choices deterministically from u, and to multiply factors for all observations and those random choices in F_Q. To compute the denominator, we can execute program P, drawing random choices from t, and accumulate factors analogously.

5.3 Trace Translator Error

Given a correspondence f for programs P and Q, we analyze trace translator $\mathcal{R} = (P, Q, k_{P\to Q}, \ell_{Q\to P})$ with $k_{P\to Q}$ defined by Equation (6) and $\ell_{Q\to P}$ defined by Equation (7). Let $Q^{(f)}$ denote the distribution produced by sampling $u \sim Q$ and then only keeping random choices that have a correspondence in f. Let $P^{(f)}$ denote the distribution produced by sampling $u \sim \eta_{P\to Q}$ and then only keeping random choices that have a correspondence in f. Note that the random choices in $\eta_{P\to Q}$ that have a correspondence in f follow the same distribution as in P, because $k_{P\to Q}$ does not modify those random choices. Samples from $Q^{(f)}$ and $P^{(f)}$ are *partial traces* of program Q—they only contain values for random choices that are in the correspondence f. We denote a partial trace of program Q by s. Let $Q(u|s)$ denote the posterior distribution Q on traces u, conditioned on u being consistent with a partial trace s:

$$Q(u|s) \propto \Pr[u \sim Q] \prod_{i\in R_u\cap R_s} \delta(u_i, s_i)$$

Likewise, let $\eta_{P\to Q}(u|s)$ denote the distribution $\eta_{P\to Q}(u)$ on traces u, conditioned on u being consistent with s. Note that $\eta_{P\to Q}(u|s)$ is precisely the probability that the forward kernel produces a trace u from a previous trace t that agrees with s on the corresponding random choices:

$$\eta_{P\to Q}(u|s) = k_{P\to Q}(u;t) \text{ for any } t \text{ s.t. } s_i = t_{f(i)} \forall i \in R_u \cap R_s$$

Let $r = f[s]$ denote the partial trace of P with $r_{f(i)} = s_i$ for each i in s. Define $P(t|r)$ analogously to $Q(u|s)$, and define $\eta_{Q\to P}(t|r)$ analogously to $\eta_{P\to Q}(u|s)$. Then, the trace translator error is a sum of three non-negative error terms:

$$\begin{aligned}\epsilon(\mathcal{R}) =& D_{KL}\left(Q^{(f)} \,\middle\|\, P^{(f)}\right) \\ &+ \mathbb{E}_{s\sim Q^{(f)}}\left[D_{KL}\left(Q(\cdot|s) \,\middle\|\, \eta_{P\to Q}(\cdot|s)\right)\right] \\ &+ \mathbb{E}_{s\sim Q^{(f)}}\left[D_{KL}\left(\eta_{Q\to P}(\cdot|f[s]) \,\middle\|\, P(\cdot|f[s])\right)\right]\end{aligned}$$

The first term reflects the difference in the probabilistic semantics of the corresponding random choices under the two programs, where $Q^{(f)}$ defines the semantics under Q and $P^{(f)}$ defines the semantics under P. The second term reflects the error introduced by sampling non-corresponding random choices in Q by evaluation (e.g. sampling from the 'prior') as in $\eta_{P\to Q}(\cdot|s)$, instead of by posterior sampling as in $Q(\cdot|s)$. The third term accounts for the error in the estimate of the weight, which is determined by the error of sampling non-corresponding random choices in P by evaluation ($\eta_{Q\to P}(\cdot|f[s])$), instead of by posterior sampling ($P(\cdot|f[s])$). Note that if every random choice in P is in correspondence with some random choice in Q, then the third term is zero. Also note that if all random choices in both programs are in the correspondence f, then $\epsilon(\mathcal{R})$ reduces to the difference in probabilistic semantics $D_{KL}(Q^{(f)}||P^{(f)})$.

5.4 Correspondence for Loops

When we add loops to the probabilistic programming language in Section 3, the same expression may be executed multiple times. To uniquely identify random choices, we index them both by their syntactic position and by their (possibly nested) loop index, as in [44]. This allows us to introduce correspondence between random choices in P and in Q.

As a simple example, consider the geometric distribution implemented in Figure 6, that counts the number of Bernoulli trials until a failure. Suppose the Bernoulli choices in the

condition are indexed with $\mathbb{N} = \{1, 2, \dots\}$. When changing the success probability from $p = \frac{1}{2}$ to $p = \frac{1}{3}$, we can use the correspondence f that maps i to i for $i \in \mathbb{N}$.

Alternatively, we can handle bounded loops by loop unrolling. Since the loop in Figure 6 is unbounded, this is not possible here. Often however, loop unrolling is sufficient, e.g. many machine learning applications loop over a finite number of data points, finite number of dimensions, or a number of latent components.

```
p=1/2;
n=1;
while(flip(p))
  n++;
```

Figure 6. Geometric distribution.

6 Correspondence from Program Edits

If Q is the result of a small edit to the text of P, most parts of P will not be affected by that edit. Then, a simple correspondence between random choices in P and Q can be constructed automatically. In this case, the translation can be optimized due to potentially massive cancellation in the weight expression (8).

We generate a semantic correspondence automatically from a program edit by assuming that random expressions that correspond *syntactically* in the two programs also correspond *semantically*. Concretely, for two random expressions in syntactic correspondence, we place their random choices in (semantic) correspondence.

Note that the syntactic correspondence is likely to, but need not, result in a good semantic correspondence—the jump from syntax to semantics is best viewed as an informed heuristic. The algorithm guarantees soundness (as in Lemma 2), even if the syntactic correspondence does not result in a good semantic correspondence. In this case however, the convergence in Lemma 2 will be slower, requiring more samples to achieve a certain error.

Partial Execution of Q When Q is the result of an edit to P, the procedure of sampling from the forward kernel (Section 5.1) and evaluation of the weight estimate (Equation 8) can be optimized to avoid a full execution of program Q.

Consider the trace $t = [b \mapsto 1, c \mapsto 4, d \mapsto 0]$ for the original program depicted in Figure 7. We assume that t is provided to the algorithm in the form of a graph data structure $\mathcal{G}_t$, where every expression, sub-expression, and statement evaluated during the construction of t is a node. Figure 7 depicts the graph $\mathcal{G}_t$ on the left. For convenience, we conflate assignments $x = e$ with the node for their right-hand side e. We introduce an edge between nodes n_1 and n_2 of $\mathcal{G}_t$ whenever n_1 must be evaluated in order to evaluate node n_2. Note that the graph data structure is similar to the *probabilistic execution trace* introduced in the Venture platform [29].

Given $\mathcal{G}_t$ and an edit to P that results in Q, we construct a trace u of Q, its graph $\mathcal{G}_u$, and the weight estimate $\hat{w}_{P\to Q}(u; t)$, by first deleting nodes in $\mathcal{G}_t$ whose source code in P was part of the edit, adding new nodes for any expressions and statements introduced in the edit, and then propagating changes from these nodes throughout the dependency graph in topological order (i.e. triggering re-evaluation of nodes if any of their parents in the graph were re-evaluated).

To traverse the changed nodes in topological order, we first compute a forward slice of the dependency graph that contains an overapproximation of all the nodes that should be re-evaluated during the current propagation of changes. Then, we order the resulting set of nodes topologically. One could instead maintain the global order of nodes explicitly using a dedicated data structure. Then, one would use a priority queue to dynamically process the set of re-evaluated nodes in topological order. This technique has previously been used for (deterministic) self-adjusting computation [1].

Figure 7 displays a dependency graph $\mathcal{G}_u$ constructed based on the edit from $a = 1$ to $a = 2$. In constructing $\mathcal{G}_u$, we removed node $a = 1$ and replaced it by $a = 2$. We then propagated that change through the dependency graph. Note that the change does not propagate through node $b =$ **flip**$(a/3)$, because the correspondence allows one to reuse the random choice $b \mapsto 1$ in the translated trace u. Because u takes the then-branch instead of the else-branch, node $c =$ **uniform**$(0, 5)$ and its parents must be deleted, and replaced by those in the else-branch.

The propagation amounts to a partial execution of Q because nodes for random choices that were part of the correspondence f do not trigger re-evaluation of their children.

Efficient Weight Estimate Evaluation Recall the weight estimate $\hat{w}_{P\to Q}(u; t)$ in Equation 8 is defined by:

$$\frac{\prod_{i \in F_Q \cap R_u} \Pr[u_i \sim Q \mid u_{1:i-1}] \prod_{i \in O_u} \Pr[i \sim Q \mid u_{1:i-1}]}{\prod_{i \in F_P \cap R_t} \Pr[t_i \sim P \mid t_{1:i-1}] \prod_{i \in O_t} \Pr[i \sim P \mid t_{1:i-1}]}$$

To efficiently compute $\hat{w}_{P\to Q}(u; t)$, we initialize it to 1 prior to propagation. For each random choice i of Q that has a correspondence $f(i)$ and is visited during propagation, the factors $\Pr[u_i \sim Q \mid u_{1:i-1}]$ and $\Pr\left[t_{f(i)} \sim P \mid t_{1:f(i)-1}\right]$ are evaluated and factored into the numerator and denominator of the weight estimate, respectively. Note that these probabilities can be computed from the values of the random choices and their arguments.

To efficiently deal with observations, we require a correspondence analogous to the correspondence for random choices. For each observation i of Q that is visited during propagation, we compute $\Pr[i \sim Q \mid u_{1:i-1}]$ and factor this into the numerator of the weight estimate. If i has a corresponding observation in t, we factor $\Pr\left[f(i) \sim P \mid t_{1:f(i)-1}\right]$ into the denominator. For any observation i of P that was removed from the graph (because there was no corresponding observation in u), we compute $\Pr[i \sim P \mid t_{1:i-1}]$ and factor this into the denominator of the weight estimate. If we were to add **observe**(**flip**$(1/5) == d$) to the modified program in

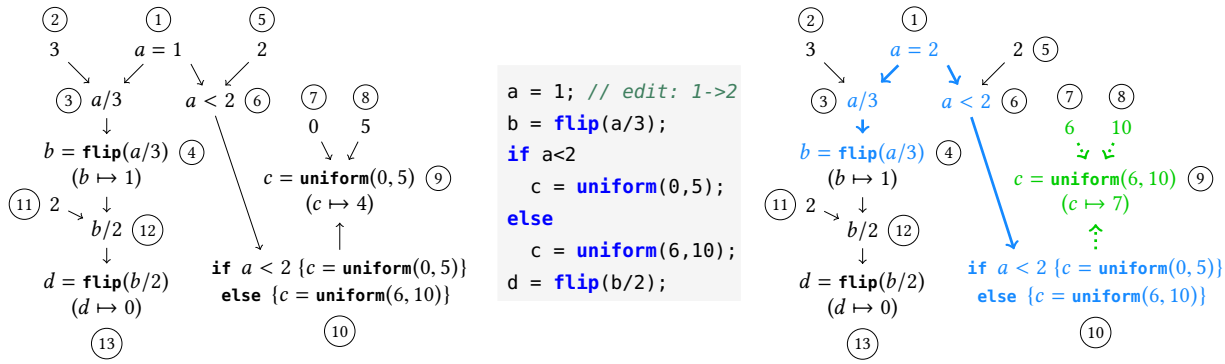

Figure 7. Representation of two traces of two programs that are related by a change of a constant ($a = 1$ vs $a = 2$). On the left, we show the trace of the original program. Each node corresponds to an expression, sub-expression or statement evaluated during the construction of the trace. Circled numbers indicate the order of evaluation. On the right, we show the trace of the modified program. Edges traversed during the construction of the modified trace are bold. Edges from newly created nodes are dotted.

Figure 7, the observation would be visited during propagation and hence factored into the numerator of the weight estimate. If $\texttt{observe}(\texttt{flip}(1/5) == d)$ would occur in both the original and the modified program, it would not lead to a factor (neither in the nominator nor the denominator), because it is not visited during propagation.

Note that all factors in the numerator of Equation (8) that were not factored into the weight estimate (i.e. factors for corresponding random choices, or observations, whose arguments were not changed from P to Q) cancel with identical factors in the denominator.

7 Implementation and Evaluation

In this section, we describe two implementations of our approach. Their evaluation shows that using incremental inference can improve the efficiency of inference by orders of magnitude.

7.1 Implementation

To illustrate the generality of our approach, we implemented our algorithms and evaluated them in two separate probabilistic programming systems. The incremental inference approach described in Section 4 and Section 5 is compatible with embedded probabilistic languages implemented using the lightweight transformational compilation design of [44]. Runtime systems for languages following this design run the program end-to-end and score each random choice. The optimized algorithm for efficient trace translation that can be used when the two programs are related by a small edit (Section 6), relies on a more involved runtime that tracks dependencies between random choices in a trace, allowing to run the modified program only partially.

For a Lightweight Embedded Language We implemented a lightweight embedded probabilistic programming language, embedded in Julia, based on the transformational compilation design of [44], with built-in support for trace translation. In our language, the programmer may annotate random choices with 'addresses' that may be dynamically computed. In our implementation of incremental inference for this lightweight language, to specify a semantic correspondence required by the algorithm of Section 5, the user defines a Julia function that maps addresses of random choices in the second program Q to addresses in the first program P. In this language, observations are not specified in the probabilistic program itself, but instead are represented externally by constraints on the values of random choices at certain addresses.

For a Language with Dependency Tracking We also implemented our approach in PSI [17], a probabilistic programming language for exact symbolic and approximate sampling-based inference, that includes a dependency tracking mechanism suitable for implementing the algorithm of Section 6.

7.2 Robust Bayesian Linear Regression

We first evaluated our lightweight incremental inference implementation on a data set of hospital operating costs and quality measures for 305 municipalities in the United States [43]. We modeled the data using a Bayesian regression probabilistic program P (shown in Listing 1 in the supplementary material). We then wrote a variant of P, denoted Q, that was modified to be more robust to outliers (shown in Listing 2). Specifically, Q allows for the possibility that a data point is an outlier, and contains a random choice not present in P that defines the variance of outliers. Robust Bayesian regression is a practical family of statistical models that generally lack closed-form posterior solutions [5]. We evaluated the incremental inference implementation, as well as an implementation of (non-incremental) MCMC inference, for the task of estimating the posterior mean of the slope parameter

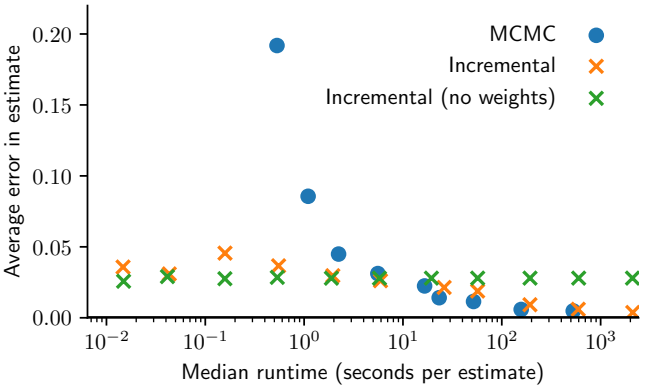

Figure 8. Evaluation of incremental inference (Algorithm 2) and MCMC for parameter estimation in probabilistic for robust regression, applied to a data set of hospital operating costs and quality metrics.

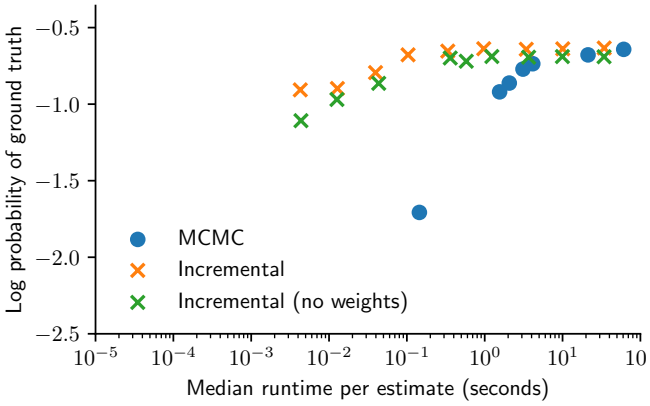

Figure 9. Evaluation of incremental inference (Algorithm 2) and MCMC for posterior inference over hidden states in a higher-order HMM, applied to typo correction.

in the robust model. Both algorithms were implemented in the same lightweight probabilistic programming language.

Because exact posterior sampling is tractable in P, we use posterior samples for P as input to the incremental inference algorithm. We placed the coefficients of the regression (the intercept and slope) in correspondence for the transition from P to Q. The incremental algorithm did not use MCMC updates after trace translation. The MCMC algorithm was based on a cycle of independent Metropolis updates to each latent variable in Q. We also evaluated a variant of the incremental algorithm that does not utilize the weight estimates produced by the trace translator. Figure 8 shows the results, which show that incremental inference gives a substantial 84% reduction in mean error for approximately 10x less runtime, relative to the MCMC approach, using a hand-optimized MCMC algorithm as the gold-standard (incremental inference gave 0.031 error at 0.043 seconds per estimate, and MCMC gave 0.19 error at 0.53 seconds per estimate). Both incremental inference and MCMC converged to the correct value, but incremental inference without use of the weight estimates did not.

7.3 Higher-order Markov Model

We next evaluated the lightweight incremental inference implementation on a typo correction task. We trained a first-order hidden Markov model and a second-order hidden Markov model on a training set of 29,056 words with typos and associated ground truth, and wrote probabilistic programs P and Q for the first-order and second-order models, respectively, shown in Listing 3 and Listing 4. The presence of second-order dependencies in Q impedes exact inference, whereas exact samples from the first-order model are efficiently obtained using dynamic programming. We use exact posterior samples for P as input to the incremental inference algorithm. We placed each hidden state in correspondence for the transition from P to Q (note that there are no other latent random choices in either P or Q). The incremental algorithm did not use MCMC updates. We also evaluated a Gibbs sampling algorithm for Q implemented in the same lightweight probabilistic programming language. We quantified the accuracy of an inference algorithm by the estimated log probability of the ground truth hidden sequence under the approximate posterior, for a set of held out words. Figure 9 shows the results. Incremental inference using 30 traces gave average per-character ground truth posterior probability of 0.41 on a test set, for a median runtime of 0.013 seconds, whereas the Gibbs sampler for the second-order HMM performed substantially worse, giving 0.18 posterior probability, for approximately 10x the runtime (0.14 seconds, for 10 back-and-forth Gibbs sweeps). Incremental inference without use of the weights, which converges to the posterior of the first-order model P and not the posterior of the second-order model Q, performed worse than incremental inference using the weights, giving 0.38 posterior probability in 0.14 seconds with 30 traces.

7.4 Gaussian Mixture Model

To evaluate the optimized trace translation algorithm described in Section 6, we implemented a Gaussian mixture model in PSI (code shown in Listing 5). We consider an edit to the program that modifies the value of a hyperparameter—the variance of the prior on cluster centers. When the mixture model models N data points from K clusters, the algorithm of Section 5 scales as $O(N + K)$ because it visits every element of the trace, but the algorithm of Section 6 scales as $O(K)$, because the changed hyperparameter only affects the cluster centers, which are in correspondence between the two programs, and do not propagate the change forward. The results are depicted in Figure 10.

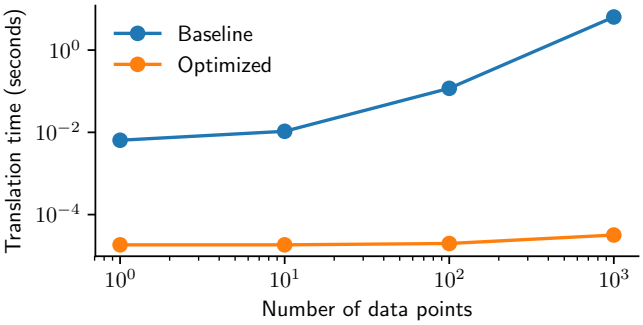

Figure 10. Comparing runtime of the baseline trace translation algorithm (Section 5) against the optimized algorithm that uses dependency tracking (Section 6), as the number of data points (and therefore the size of the trace) grows.

8 Related Work

Although Sequential Monte Carlo (SMC) has been previously applied for inference in probabilistic programs, existing work has focused on using SMC for specialized classes of incrementalization: Using sequential observation of data to make inference more efficient [19, 29, 37, 45] or iteratively refining the domain of random choices [41]. Instead, we provide a general framework for incremental inference that allows reuse of inference computation between two arbitrary programs, provided that a semantic correspondence is given between their random choices. Our work generalizes the sequential observation case studied in previous work. For example, we can also handle adding or removing latent variables.

The adaptation of approximate samples from one distribution to another is the basis of importance sampling, which has been used in Bayesian statistics to update posterior approximations in light of new data or changed priors [40]. Solving a sequence of inference problems constructed by incremental modification of a model is often used instrumentally in statistics as a means of solving the final inference problem more efficiently [9, 13, 34], and a similar approach is used in simulated annealing for optimization [23]. Our approach adopts and builds on the SMC formalism of [13] to propose a general approach for incremental inference in probabilistic programs.

The problem of exact inference when incrementally modifying factor graphs and Bayesian networks has been studied in [2–4, 14, 15, 27]. Instead, we tackle incremental approximate Monte Carlo inference, which has very different scaling properties to these approaches—our approach scales primarily in the semantic difference induced by the change, instead of in 'syntactic' parameters like the number of added random variables. For *deterministic* programs, Acar [1] describes self-adjusting computation that adapts an existing program trace to account for modified input. Partush [38] investigates possible relationships between variables in two deterministic programs, with the goal of computing semantic differences between the two programs. Investigating different correspondences of random choices in the original and the modified program is a possible extension of our work.

Incremental data flow analysis of probabilistic programs (with only discrete and bounded random choices) has been investigated in [48]. Also, various probabilistic programming systems perform incremental computation for efficient evaluation of Metropolis-Hastings acceptance ratios [24, 29, 39, 46, 47]. These systems efficiently compute the ratio of probabilities of an original trace and a proposal trace of the same program, whereas our efficient weight computation procedure (Section 6) compares probabilities of traces from two different programs.

9 Conclusion

Approximate inference for a program in a general-purpose probabilistic programming language is an intrinsically hard problem. However, this task may be substantially simplified if we are given the inference results for a related program.

We have presented the concept of a trace translator that adapts traces of an original program into traces of a modified program. By formulating this concept as a general framework in terms of SMC, we have provided formal guarantees that will be useful for future work in this domain. Given a semantic correspondence between random choices of two programs, we provide a trace translator that samples from the modified program by picking the values of random choices based on the values of corresponding random choices in the original program. When two programs are related by an edit, we can infer a semantic correspondence based on syntactic correspondence, and we can perform trace translation asymptotically more efficiently using partial edits of a dependency graph data structure. Note that we make use of two distinct forms of incremental computation: (i) reusing inference results from a program with similar probabilistic semantics, and (ii) incrementally transforming a trace data structure from one program to another. The first depends on the (probabilistic) semantic relationship between the programs, and the second relies on their syntactic relationship.

We have implemented and evaluated our approach, demonstrating that it can improve the efficiency of inference by orders of magnitude by leveraging inference for a simpler model. However, the efficiency gains depend critically on the *trace translator error* (Section 5.3), which is smaller when corresponding random choices follow a similar distribution, and increases with each additional non-corresponding random choice. Reducing the error of the trace translator by exploiting analytically tractable conditional distributions for non-corresponding choices is a promising area for future work. Also, finding a good correspondence can be hard in general. Thus, techniques for automatically identifying correspondences between general programs is an interesting area of future work.

Acknowledgments

This research was supported in part by the US Department of the Air Force contract FA8750-17-C-0239, grants from the MIT Media Lab/Harvard Berkman Center Ethics & Governance of AI Fund and the MIT CSAIL Systems that Learn Consortium, a gift from the Aphorism Foundation, and the the US Department of Defense through the the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

References

[1] Umut A Acar. 2009. Self-adjusting computation (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 1–6.

[2] Umut A Acar, Alexander T Ihler, Ramgopal Mettu, and Özgür Sümer. 2012. Adaptive inference on general graphical models. *arXiv preprint arXiv:1206.3234* (2012).

[3] Umut A Acar, Alexander T Ihler, Ramgopal R Mettu, and Özgür Sümer. 2007. Adaptive Bayesian inference. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 1441–1448.

[4] Hamza Agli, Philippe Bonnard, Christophe Gonzales, and Pierre-Henri Wuillemin. 2016. Incremental junction tree inference. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Springer, 326–337.

[5] James O Berger, Elías Moreno, Luis Raul Pericchi, M Jesús Bayarri, José M Bernardo, Juan A Cano, Julián De la Horra, Jacinto Martín, David Ríos-Insúa, Bruno Betrò, et al. 1994. An overview of robust Bayesian analysis. *Test* 3, 1 (1994), 5–124.

[6] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017).

[7] Arun Chaganty, Aditya Nori, and Sriram Rajamani. 2013. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*. 153–160.

[8] Sourav Chatterjee and Persi Diaconis. 2015. The sample size required in importance sampling. *arXiv preprint arXiv:1511.01437* (2015).

[9] Nicolas Chopin. 2002. A sequential particle filter method for static models. *Biometrika* 89, 3 (2002), 539–552.

[10] Gregory F Cooper. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial intelligence* 42, 2-3 (1990), 393–405.

[11] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. 2017. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977* (2017).

[12] Paul Dagum and Michael Luby. 1993. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial intelligence* 60, 1 (1993), 141–153.

[13] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential monte carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (2006), 411–436.

[14] M Julia Flores, José A Gámez, and Kristian G Olesen. 2002. Incremental compilation of Bayesian networks. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 233–240.

[15] M Julia Flores, Jose A Gámez, and Kristian G Olesen. 2011. Incremental compilation of bayesian networks based on maximal prime subgraphs. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 19, 02 (2011), 155–191.

[16] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic netkat. In *European Symposium on Programming Languages and Systems*. Springer, 282–309.

[17] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

[18] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

[19] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2017-8-26.

[20] Roger Grosse, Ruslan R Salakhutdinov, William T Freeman, and Joshua B Tenenbaum. 2012. Exploiting compositionality to explore a large space of model structures. *arXiv preprint arXiv:1210.4856* (2012).

[21] Jonathan H Huggins and Daniel M Roy. 2015. Convergence of sequential Monte Carlo-based sampling methods. *arXiv preprint arXiv:1503.00966* (2015).

[22] Jin H Kim and Judea Pearl. 1983. A computational model for causal and diagnostic reasoning in inference systems.. In *IJCAI*, Vol. 83. 190–193.

[23] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. 1987. Optimization by simulated annealing. In *Spin Glass Theory and Beyond: An Introduction to the Replica Method and Its Applications*. World Scientific, 339–348.

[24] Oleg Kiselyov. 2016. Probabilistic programming language and its incremental evaluation. In *Asian Symposium on Programming Languages and Systems*. Springer, 357–376.

[25] Martin Kučera, Petar Tsankov, Timon Gehr, Marco Guarnieri, and Martin Vechev. 2017. Synthesis of Probabilistic Privacy Enforcement. *Analysis* 1 (2017), 1.

[26] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. 2016. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900* (2016).

[27] Wei Li, Peter Van Beek, and Pascal Poupart. 2006. Performing incremental Bayesian inference by dynamic model counting. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 21. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 1173.

[28] Jun S Liu and Rong Chen. 1995. Blind deconvolution via sequential imputations. *J. Amer. Statist. Assoc.* 90, 430 (1995), 567–576.

[29] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099* (2014).

[30] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. The MIT Press.

[31] Lawrence M Murray. 2013. Bayesian state-space modelling on high-performance hardware using LibBi. *arXiv preprint arXiv:1306.3277* (2013).

[32] Lawrence M Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B Schön. 2017. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. *arXiv preprint arXiv:1708.07787* (2017).

[33] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming*. Springer, 62–79.

[34] Radford M Neal. 2001. Annealed importance sampling. *Statistics and computing* 11, 2 (2001), 125–139.

[35] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs.

[36] Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 208–217.

[37] Brooks Paige and Frank Wood. 2014. A compilation target for probabilistic programming languages. *arXiv preprint arXiv:1403.0504* (2014).

[38] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. *ACM SIGPLAN Notices* 49, 10 (2014), 811–828.

[39] Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. 2016. C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*. 28–37.

[40] Adrian FM Smith and Alan E Gelfand. 1992. Bayesian statistics without tears: a sampling–resampling perspective. *The American Statistician* 46, 2 (1992), 84–88.

[41] Andreas Stuhlmüller, Robert XD Hawkins, N Siddharth, and Noah D Goodman. 2015. Coarse-to-fine sequential monte carlo for probabilistic programs. *arXiv preprint arXiv:1509.02962* (2015).

[42] Sebastian Thrun. 2002. Probabilistic robotics. *Commun. ACM* 45, 3 (2002), 52–57.

[43] John E. Wennberg, Elliott S. Fisher, David C. Goodman, and Jonathan S. Skinner. 2008. Tracking the Care of Patients with Severe Chronic Illness - The Dartmouth Atlas of Health Care 2008.

[44] David Wingate, Andreas Stuhlmueller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 770–778.

[45] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*. 1024–1032.

[46] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242* (2016).

[47] Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. 2014. Generating efficient MCMC kernels from probabilistic programs. In *Artificial Intelligence and Statistics*. 1068–1076.

[48] Jieyuan Zhang, Yulei Sui, and Jingling Xue. 2017. Incremental Analysis for Probabilistic Programs. In *International Static Analysis Symposium*. Springer, 450–472.

Supplemental materials

A Guarantees of SMC

In this section, we derive some guarantees of SMC. These are adaptations of standard results to our setting. In the following, the expectation and probabilities are over T, U sampled by $T \sim P$ and $U \sim k_{P\to Q}(\cdot; T)$.

First, we show that in expectation, the weight estimate $\hat{w}_{P\to Q}(u;t)$ introduced in Section 4 is proportional to the importance weight $w_{P\to Q}(u)$.

Lemma 4.

$$\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T) \mid U = u] = \frac{Z_Q}{Z_P} w_{P\to Q}(u) \propto w_{P\to Q}(u)$$

Proof.

$$\begin{aligned}
&\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T) \mid U = u] \\
&= \sum_{t\in\mathcal{T}_P} \Pr_{T,U}[T = t \mid U = u]\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T) \mid U = u, T = t] \\
&= \sum_{t\in\mathcal{T}_P} \Pr_{T,U}[T = t \mid U = u]\hat{w}_{P\to Q}(u;t) \\
&= \sum_{t\in\mathcal{T}_P} \frac{\Pr_{T,U}[T = t, U = u]}{\Pr_{T,U}[U = u]}\hat{w}_{P\to Q}(u;t) \\
&= \sum_{t\in\mathcal{T}_P} \frac{\Pr[t \sim P]k_{P\to Q}(u,t)}{\sum_{s\in\mathcal{T}_P}\Pr[s \sim P]k_{P\to Q}(u;s)}\hat{w}_{P\to Q}(u;t) \\
&= \sum_{t\in\mathcal{T}_P} \frac{\Pr[t \sim P]k_{P\to Q}(u,t)}{\sum_{s\in\mathcal{T}_P}\Pr[s \sim P]k_{P\to Q}(u;s)} \frac{\widetilde{\Pr}[u \sim Q]\ell_{Q\to P}(t;u)}{\widetilde{\Pr}[t \sim P]k_{P\to Q}(u;t)} \\
&= \sum_{t\in\mathcal{T}_P} \frac{\Pr[t \sim P]k_{P\to Q}(u,t)}{\widetilde{\Pr}[t \sim P]k_{P\to Q}(u;t)} \frac{\widetilde{\Pr}[u \sim Q]\ell_{Q\to P}(t;u)}{\sum_{s\in\mathcal{T}_P}\Pr[s \sim P]k_{P\to Q}(u;s)} \\
&= \frac{Z_Q}{Z_P} \sum_{t\in\mathcal{T}_P} \frac{\Pr[u \sim Q]\ell_{Q\to P}(t;u)}{\sum_{s\in\mathcal{T}_P}\Pr[s \sim P]k_{P\to Q}(u;s)} \\
&= \frac{Z_Q}{Z_P} \frac{\Pr[u \sim Q]}{\sum_{s\in\mathcal{T}_P}\Pr[s \sim P]k_{P\to Q}(u;s)} \\
&= \frac{Z_Q}{Z_P} \frac{\Pr[u \sim Q]}{\eta(u)} \\
&= \frac{Z_Q}{Z_P} w_{P\to Q}(u)
\end{aligned}$$

□

Lemma 4 is important because it allows to compute the expectation of any test function $\varphi : \mathcal{T}_Q \to \mathbb{R}$ from traces of Q to real numbers.

Lemma 5. *Let $\varphi : \mathcal{T}_Q \to \mathbb{R}$ by any test function from traces of Q to real numbers. Then,*

$$\mathbb{E}_{U,T}[\hat{w}_{P\to Q}(U;T)\varphi(U)] = \frac{Z_Q}{Z_P}\mathbb{E}_{U\sim Q}[\varphi(U)]$$

Proof.

$$\begin{aligned}
&\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T)\varphi(U)] \\
&= \sum_{u\in\mathcal{T}_Q} \Pr_U[U = u]\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T)\varphi(U) \mid U = u] \\
&= \sum_{u\in\mathcal{T}_Q} \Pr_U[U = u]\varphi(u)\mathbb{E}_{T,U}[\hat{w}_{P\to Q}(U;T) \mid U = u] \\
&= \sum_{u\in\mathcal{T}_Q} \Pr_U[U = u]\varphi(u)\frac{Z_Q}{Z_P} w_{P\to Q}(u) \\
&= \sum_{u\in\mathcal{T}_Q} \Pr_U[U = u]\varphi(u)\frac{Z_Q}{Z_P}\frac{\Pr[u \sim Q]}{\eta(u)} \\
&= \frac{Z_Q}{Z_P}\sum_{u\in\mathcal{T}_Q} \varphi(u)\Pr[u \sim Q] \\
&= \frac{Z_Q}{Z_P}\mathbb{E}_{U\sim Q}[\varphi(U)]
\end{aligned}$$

Here, we have used that $\eta(u) = \Pr[U = u]$. □

Lemma 6. *Let $(P, Q, k_{P\to Q}, \ell_{Q\to P})$ be a trace translator. Let $t_j \sim P$ and $u_j \sim k_{P\to Q}(\cdot, t_j)$ for $j \in \{1, \ldots, M\}$. Then, almost surely:*

$$\frac{1}{M}\sum_{j=1}^{M} \hat{w}_{P\to Q}(u_j; t_j) \xrightarrow[M\to\infty]{} \frac{Z_Q}{Z_P}$$

Proof. By Lemma 5, we have for $\varphi(u) \equiv 1$

$$\mathbb{E}_{U,T}[\hat{w}_{P\to Q}(U;T) \cdot 1] = \frac{Z_Q}{Z_P} \cdot 1$$

By the law of large numbers, we get

$$\frac{1}{M}\sum_{j=1}^{M} \hat{w}_{P\to Q}(u_j; t_j) \xrightarrow[M\to\infty]{} \frac{Z_Q}{Z_P}$$

□

The previous Lemmas allow to approximate the expectation of any test function $\varphi : \mathcal{T}_Q \to \mathbb{R}$ from traces of Q to real numbers.

Lemma 7. *Let $(P, Q, k_{P\to Q}, \ell_{Q\to P})$ be a trace translator. Let $\varphi : \mathcal{T}_Q \to \mathbb{R}$ be any test function.*

Let $t_j \sim P$ and $u_j \sim k_{P\to Q}(\cdot, t_j)$ for $j \in \{1, \ldots, M\}$. Then, almost surely,

$$\frac{\sum_{j=1}^{M} \hat{w}_{P\to Q}(u_j; t_j)\varphi(u_j)}{\sum_{j=1}^{M} \hat{w}_{P\to Q}(u_j; t_j)} \xrightarrow[M\to\infty]{} \mathbb{E}_{U\sim Q}[\varphi(U)]$$

Proof. From Lemma 5, we have by the law of large numbers:

$$\frac{1}{M}\sum_{j=1}^{M} \hat{w}_{P\to Q}(u_j; t_j)\varphi(u_j) \xrightarrow[M\to\infty]{} \frac{Z_Q}{Z_P}\mathbb{E}_{U\sim Q}[\varphi(U)]$$

Lemma 7 follows by combining this with Lemma 6. □

B Scaling of Necessary Sample Size

Applying Theorem 1.2 of [8] with the proposal distribution $\Pr[t \sim P]\, k_{P\to Q}(u;t)$ and with the target distribution $\Pr[u \sim Q]\, \ell_{Q\to P}(t;u)$, shows that the necessary and sufficient sample size is approximately exponential in the KL divergence from the target to the proposal, which is $\epsilon(R)$.

C Evaluation Programs

In this section, we provide the probabilistic programs used for evaluation.

```
@probabilistic function (params::NoOutlierModelParams,
                         x::Vector{Float64})
  slope = @address(normal(0., params.prior_std),
                   ADDR_SLOPE)
  intercept = @address(normal(0., params.prior_std),
                       ADDR_INTERCEPT)
  ys = Vector{Float64}(length(x))
  for (i, xi) in enumerate(x)
    y_mean = intercept + slope * xi
    ys[i] = @address(normal(y_mean, params.std),
                     addr_y(i))
  end
end
```

Listing 1. Bayesian linear regression

```
@probabilistic function (params::OutlierModelParams,
                         x::Vector{Float64})
  prob_outlier = params.prob_outlier
  inlier_std = params.inlier_std
  outlier_log_var = @address(
      normal(params.outlier_log_var_mu,
             params.outlier_log_var_std),
      ADDR_OUTLIER_LOG_VAR)
  outlier_std = sqrt(exp(outlier_log_var))
  slope = @address(normal(0., params.prior_std),
                   ADDR_SLOPE)
  intercept = @address(normal(0., params.prior_std),
                       ADDR_INTERCEPT)
  ys = Vector{Float64}(length(x))
  for (i, xi) in enumerate(x)
    y_mean = intercept + slope * xi
    ys[i] = @address(two_normals(y_mean, prob_outlier,
                                 inlier_std, outlier_std),
                     linreg_addr_y(i))
  end
end
```

Listing 2. Robust Bayesian linear regression

```
@probabilistic function (params::FirstOrderParams,
                         num_steps::Int)
  x = Vector{Int}(num_steps)
  if num_steps >= 1
    x[1] = @address(uniform_discrete(1,  params.num_states),
                    addr_hidden(1))
  end
  for i=2:num_steps
    x[i] = @address(categorical_log(
                      params.log_transition_model[x[i-1],:]),
                    addr_hidden(i))
  end
  for i=1:num_steps
    @address(categorical_log(
               params.log_observation_model[x[i],:]),
             addr_y(i))
  end
end
```

Listing 3. First-order hidden Markov model

```
@probabilistic function (params::SecondOrderParams,
                         num_steps::Int)
  x = Vector{Int}(num_steps)
  if num_steps >= 1
    x[1] = @address(uniform_discrete(1, params.num_states),
                    addr_hidden(1))
  end
  if num_steps >= 2
    x[2] = @address(categorical_log(
             params.log_first_order_transition_model[x[1],:]),
            addr_hidden(2))
  end
  for i=3:num_steps
    x[i] = @address(categorical_log(
             params.log_transition_model[x[i-2],x[i-1],:]),
            addr_hidden(i))
  end
  for i=1:num_steps
    @address(categorical_log(
               params.log_observation_model[x[i],:]),
             addr_y(i))
  end
end
```

Listing 4. Second-order hidden Markov model

```
def main(sigma,n){
  k := 10;
  centers := array(k,0);
  for i in [0..k){
    centers[i]=gauss(0,sigma);
  }
  data := array(n,0);
  for i in [0..n){
    data[i]=gauss(centers[uniformInt(0,k-1)],1);
  }
  return data;
}
```

Listing 5. Finite Gaussian mixture model (PSI)