# Incremental Analysis for Probabilistic Programs

Jieyuan Zhang[1], Yulei Sui[1], and Jingling Xue[1,2]

[1] School of Computer Science and Engineering, UNSW Sydney, Australia
[2] Advanced Innovation Center for Imaging Technology, CNU, China
{jieyuan,ysui,jingling}@cse.unsw.edu.au

**Abstract.** This paper presents ICPP, a new data-flow-based **InC**remental analysis for **P**robabilistic **P**rograms, to infer their posterior probability distributions in response to small yet frequent changes to probabilistic knowledge, i.e., prior probability distributions and observations. Unlike incremental analyses for usual programs, which emphasize code changes, such as statement additions and deletions, ICPP focuses on changes made to probabilistic knowledge, the key feature in probabilistic programming. The novelty of ICPP lies in capturing the correlation between prior and posterior probability distributions by reasoning about the probabilistic dependence of each data-flow fact, so that any posterior probability affected by newly changed probabilistic knowledge can be incrementally updated in a sparse manner without recomputing it from scratch, thereby allowing the previously computed results to be reused. We have evaluated ICPP with a set of probabilistic programs. Our results show that ICPP is an order of magnitude faster than the state-of-the-art data-flow-based inference in analyzing probabilistic programs under small yet frequent changes to probabilistic knowledge, with an average analysis overhead of around 0.1 seconds in response to a single change.

## 1 Introduction

Uncertainty is a common feature in many modern software systems, especially statistical applications (e.g., climate change prediction, spam email filtering and ranking the skills of game players). Probabilistic programming provides a powerful approach to quantifying and characterizing the effects of these uncertainties. A Probabilistic Programming Language (PPL) usually extends an imperative language (e.g., C and Java) by adding two types of language constructs, i.e., *probabilistic assignments* for generating random values based on prior probability distributions and *observe statements* for conditioning values of variables.

Unlike an imperative program, which is mainly written for the purposes of being executed, a probabilistic program is a specification that specifies implicitly posterior probability distributions to model uncertainty of the program. Probabilistic inference is the key to reasoning about a probabilistic program by extracting explicit distributions that are implicitly specified in the program.

Generally, there are two approaches to probabilistic inference: (1) *dynamic inference*, which runs a probabilistic program a finite number of times through sampling-based Monte Carlo methods [4,7,19,25,28] and then performs inference

to calculate the statistics based on the execution traces, and (2) *static inference*, which statically computes the probability distributions without repeatedly executing the program. A typical static method [2] is to abstract a loop-free program as a probabilistic model (e.g., a Bayesian network) and then resorts to existing inference algorithms, e.g., belief propagation [24] and variational inference [35]. A recent work, DFI [8], provides more precise inference results than sampling algorithms and Bayesian modeling methods by applying data-flow analysis to analyze probabilistic programs with and without loops.

Unlike the case for imperative programs, applying data-flow analysis to infinite-state probabilistic programs is generally more expensive. Data-flow facts of probabilistic programs are probability distributions, including the values of program variables and their corresponding probabilities. Given a probabilistic program, the number of its data-flow facts depends not only on its size parameters but also the prior distributions at its probabilistic assignments and the conditions at its observe statements. As a common practice in probabilistic programming, probabilistic knowledge, which is represented by prior probability distributions and observations, is often updated under different scenarios or settings [3,6,37]. To achieve precise modeling, probabilistic assignments and observe statements are often changed in order to obtain various posterior probability distributions when writing a probabilistic program [36]. However, such small yet frequent changes affect the performance of static inference as the previous inference results become invalid once the program has been modified. Repeatedly reanalyzing a probabilistic program that undergoes small changes makes static inference costly.

Incremental analysis aims to efficiently update existing analysis results without recomputing them from scratch, allowing the previously computed information to be reused. There are a few existing works that support incremental analysis, such as pointer analysis [18,31], IDE/IFDS analysis [1], data race detection [38], symbolic execution [26], and fixed-point analysis for logic programs [14]. However, these existing techniques cannot be directly applied to analyze probabilistic programs. For probabilistic programs, frequent changes in probabilistic knowledge pose a new challenge to incremental analysis. It is still an open question as to whether we can replicate the success of previous incremental analysis for usual programs in analyzing probabilistic programs.

In this paper, we present Icpp, a new **I**n**C**remental analysis for analyzing **P**robabilistic **P**rograms, to infer its posterior probability distributions in response to small yet frequent changes to probabilistic knowledge, i.e., prior probability distributions and observations. Unlike previous incremental analyses for usual programs, which emphasize code changes, such as statement additions and deletions, Icpp focuses on changes made to probabilistic knowledge, which is the key feature in probabilistic programming.

As illustrated in Figure 1, Icpp first performs data-flow-based pre-inference. Unlike DFI [8], which explicitly computes and maintains the probability of every program state, our pre-inference generates data-flow facts with each consisting of a program state and its corresponding *probabilistic dependence*, which is used to maintain the correlation between the posterior and prior probability distri-

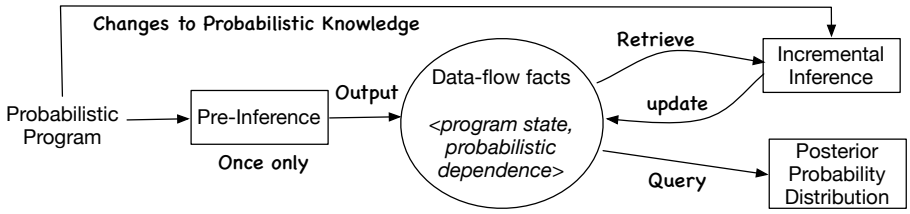


**Fig. 1.** Workflow of Icpp.

butions at probabilistic assignments. This probabilistic dependence is later retrieved to facilitate incremental inference once a change is made to a probabilistic assignment or an observe statement. Based on the dependence information, the data-flow facts are updated incrementally and propagated sparsely along the control flow to adapt to program changes, making Icpp an instantaneous incremental analysis for users to query posterior probability distributions, while achieving the same precision achieved when the program is re-analyzed entirely.

In summary, the contributions of this paper are as follows:

- We present Icpp, a new **InC**remental analysis for **P**robabilistic **P**rograms, in response to small yet frequent changes to probabilistic knowledge.
- We propose a new probabilistic dependence analysis to analyze the two distinct language constructs in probabilistic programs, probabilistic assignments and observe statements.
- We evaluate Icpp using a set of probabilistic programs from R2 [25] and DFI [8]. Our results show that Icpp is an order of magnitude faster than the state-of-the-art data-flow-based inference [8] in analyzing these programs under small yet frequent changes to probabilistic knowledge, with an average analysis overhead of around 0.1 seconds in response to a single change.

## 2 Background

In this section, we describe the preliminaries for our analysis, by focusing on the representation and inference of a probabilistic program.

### 2.1 Probabilistic Programs

Following [8,12], we represent a probabilistic program using a tiny language defined in Figure 2. This is a single-function imperative language with two added constructs: (1) a *probabilistic assignment*, $x = \texttt{Dist}(\overline{\theta})$, that assigns random values to variable $x$ based on a probability distribution $\texttt{Dist}(\overline{\theta})$, such as `Bernoulli`, `UniformInt` and `Gauss`, where $\overline{\theta}$ is a list of parameters according to a distribution model (with a continuous distribution being approximated by a discrete distribution over a finite set, following [8,21]), and (2) an *observe statement*,

| | | | |
|---|---|---|---|
| $x, y, a, b$ | $\in$ | Vars | program variables |
| $\theta$ | $\in$ | $\mathbb{R}$ | real numbers |
| `Dist` | $\in$ | {`UniformInt, Bernoulli, Gauss, ...`} | distributions |
| `uop` | ::= | $\{++, --, !\}$ | unary operations |
| `bop` | ::= | $\{+, -, \times, /, \&\&, \|\|, ==, \neq, <, >, \leq, \geq\}$ | binary operations |
| $\mathcal{E}$ | ::= | | expressions |
| | | \| x | variable |
| | | \| c | constant |
| | | \| $\mathcal{E}_1$ `bop` $\mathcal{E}_2$ | binary operation |
| | | \| `uop` $\mathcal{E}$ | unary operation |
| $\ell$ | ::= | | labeled statements |
| | | \| $x = \mathcal{E}$ | deterministic assignment |
| | | \| $\ell_1; \ell_2$ | sequential composition |
| | | \| if $\mathcal{E}$ then $\ell_1$ else $\ell_2$ | conditional composition |
| | | \| while $\mathcal{E}$ do $\ell$ | loop |
| | | \| skip | skip |
| | | \| **x = `Dist`($\overline{\theta}$)** | **probabilistic assignment** |
| | | \| **observe($\mathcal{E}$)** | **observe** |
| *Prog* | ::= | $\overline{\ell}$ | program |

**Fig. 2.** Syntax of a probabilistic program.

**observe($\mathcal{E}$)**, that conditions the expression $\mathcal{E}$ to be true. The effect of the observe statement is to block all program executions violating condition $\mathcal{E}$.

Figure 3 gives examples to illustrate the differences between an imperative program in Figure 3(a) and its probabilistic counterparts in Figures 3(b) and 3(c). Figure 3(b) replaces the deterministic assignment at line 2 in Figure 3(a) with a probabilistic assignment, so that the variable $a$ is assigned a random value based on the discrete uniform distribution `UniformInt(0,1)`, which returns one of two integers 0 and 1 with equal probability, 1/2. Figure 3(c) gives another probabilistic program by adding further `observe`(b==1) after statement $\ell_4$ in Figure 3(b) to block any execution such that $b$ is not equal to 1 at $\ell_5$.

As shown in Figure 3(d), executing the imperative program in Figure 3(a) always produces the deterministic result ($a = 0$, $b = 1$). However, probabilistic programs are nondeterministic. Executing the one in Figure 3(b) may produce one of the two different results: ($a=1, b=0$) and ($a=0, b=1$). Figure 3(e) shows a posterior distribution with equal probability 1/2 for each result. The imperative program in Figure 3(a) can be seen as a special case of the probabilistic program in Figure 3(b) with the probability of its unique deterministic result being 1.

Figure 3(f) demonstrates that the result ($a = 1$, $b = 0$) becomes infeasible with its possibility being 0 due to the condition at the observe statement. After normalization, the probability for the other result ($a = 0$, $b = 1$) becomes 1.

### 2.2 Probabilistic Inference

The key mechanism for reasoning about a probabilistic program is probabilistic inference, which explicitly calculates the posterior probability distributions implicitly specified in the program. There are two approaches: (1) *dynamic inference*, which executes programs a finite number of times through sampling-

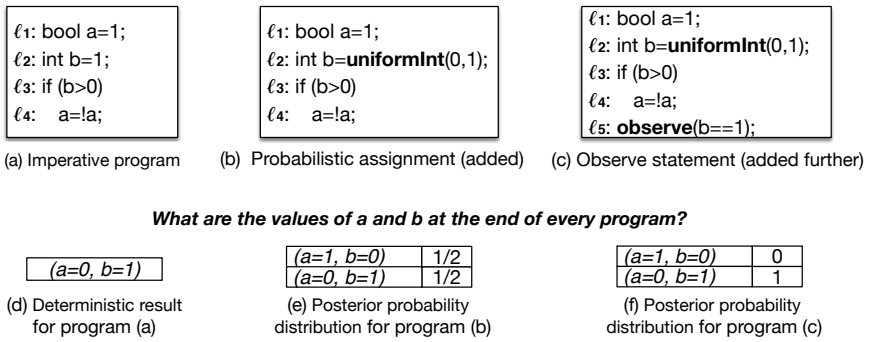


**Fig. 3.** Imperative vs. probabilistic programs.

based methods [19], such as importance sampling [11], Gibbs sampling [28] and Metropolis-Hastings sampling [7], and (2) *static inference*, which computes the probability distributions statically without running the program.

A recent data-flow-based inference, DFI [8], applies the data-flow theory for probabilistic inference by treating probability distributions as data-flow facts. DFI is path-sensitive by analyzing control-flow branch conditions. The resulting inference provides better precision than many existing methods, e.g., Expectation Propagation [22], message passing algorithm [16] and MCMC sampling [13].

In DFI, the static inference is formulated as a forward data-flow problem $(D, \sqcap, F)$. Here, $D$ represents all data-flow facts with each $\langle \sigma, \rho \rangle \in D$ consisting of a program state $\sigma$ (a set of values) and its corresponding probability $\rho$ when $\sigma$ holds. $\sqcap$ is the meet operator. $F : D \to D$ represents the set of transfer functions with $f_\ell$ being associated with node (statement) at $\ell$ in the CFG of the program.

A path-sensitive analysis computes the data-flow facts (probability distributions) by considering every executable path. We write $\pi$ to denote a path $[\ell_1, \ell_2 \ldots \ell_n]$ consisting a sequence of $n$ statements in a CFG. The transfer function for $\pi$ is $f_\pi \in F$, which is the composition of transfer functions of the first $n-1$ statements on $\pi$, i.e., $f_\pi = f_{\ell_1} \circ f_{\ell_2} \ldots f_{\ell_{n-1}}$. Note that we speak of the path $\pi$ by excluding the last statement at $\ell_n$. Finally, the set of data-flow facts, $D_{\ell n}$, that reach the beginning of a statement $\ell_n$ is computed as follows:

$$D_{\ell n} = \bigsqcap_{\pi \in paths(\ell_n)} f_\pi(\top) \tag{1}$$

where $paths(\ell_n)$ denotes the set of paths from the program entry to statement $\ell_n$ and $\top \in D$ is the standard top element in the lattice used.

When analyzing a statement $\ell$ in DFI [8], its transfer function $f_\ell$, which is defined based on the standard $Gen/Kill$ sets, is distributive, so that $f_\ell(d_1) \cup f_\ell(d_2) = f_\ell(d_1 \cup d_2)$ holds, where $d_1, d_2 \in D$. Therefore, the meet operator $\sqcap$ is the set union ($\cup$), causing the data-flow facts at a joint point to be merged, in order to reduce the number of facts propagated without affecting the precision of

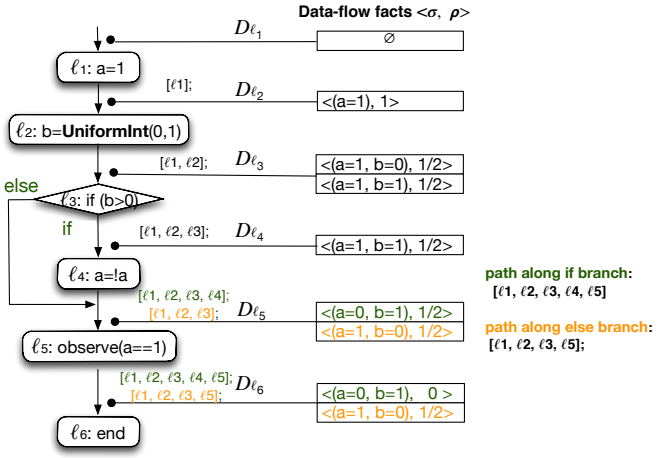


**Fig. 4.** Data-flow-based probabilistic inference.

the posterior probability distribution results. In particular, two data-flow facts $\langle \sigma_1, \rho_1 \rangle$ and $\langle \sigma_2, \rho_2 \rangle$ at a joint point are merged into $\langle \sigma_1, \rho_1 + \rho_2 \rangle$ if $\sigma_1 == \sigma_2$.

Let us take a look at the data-flow-based inference in Figure 4 by revisiting the example in Figure 3(c). After analyzing the deterministic assignment $\ell_1$, $D_{\ell_2}$ at the beginning of $\ell_2$ is $\langle (a = 1), 1 \rangle$. After analyzing the probabilistic assignment $\ell_2$, we obtain two data-flow facts representing the probability distributions for two possible states, $(a = 1, b = 0)$ and $(a = 1, b = 1)$, with their corresponding probabilities being $\rho_{(a=1,b=0)} = Pr(\ell_1 : a = 1) * Pr(\ell_2 : b = 0) = 1 * 1/2 = 1/2$ and $\rho_{(a=1,b=1)} = Pr(\ell_1 : a = 1) * Pr(\ell_2 : b = 1) = 1 * 1/2 = 1/2$, respectively.

$D_{\ell_5}$ contains the two data-flow facts reaching the beginning of $\ell_5$, $\langle (a = 0, b = 1), 1/2 \rangle$ and $\langle (a = 1, b = 0), 1/2 \rangle$, which are computed and propagated from the `if` and `else` branches, respectively. Finally, after analyzing the observe statement $\ell_5$, $D_{\ell_6}$ (without normalization) is the same as $D_{\ell_5}$ except that the probability of $(a = 0, b = 1)$ has been updated to from $1/2$ to 0.

## 3 A Motivating Example

Figure 5 gives an example to illustrate the basic idea behind ICPP when the prior probability distribution at $\ell_2$ is changed from `UniformInt`(0,1) to `UniformInt`$(-1, 1)$. This change affects the probabilities of $b$'s existing values and introduces a new value -1 to $b$. Note that observe statements are handled as a special case of probabilistic statements and will be discussed in Section 4.2.2.

Unlike DFI [8], which explicitly computes and maintains the probability $\rho$ of every state $\sigma$ reaching statement $\ell$ in terms of a data-flow fact $\langle \sigma, \rho \rangle \in D_\ell$, ICPP represents a data-flow fact in the form of $\langle \sigma, \gamma_\sigma \rangle \in D_\ell$, where $\gamma_\sigma$ is $\sigma$'s all-path probabilistic dependence (Definition 3), which implicitly represents $\sigma$'s probability $\rho_\sigma$. We obtain $\gamma_\sigma$ by merging $\sigma$'s single-path dependences $\gamma_{\pi,\sigma}$ for all

the paths $\pi$ reaching $\ell$ (Definition 2), where $\gamma_{\pi,\sigma}$ collects the probability seeds generated from the relevant probabilistic assignments on $\pi$ (Definition 1). When building a particular single-path dependence $\gamma_{\pi,\sigma}$, only one single seed is selected for a probabilistic assignment every time when it is analyzed. Therefore, a seed may appear multiple times in $\gamma_{\pi,\sigma}$ when $\pi$ contains a loop, which is handled by approximating a KL-divergence [22] between two consecutive loop iterations.

**Definition 1 (Probability Seed).** *For a probabilistic assignment $\ell : x = \texttt{Dist}(\overline{\theta})$, we define a probability seed at $\ell$ as $s = \langle \ell : x = a \rangle$, where $a$ is one of all the possible values returned by the prior distribution $\texttt{Dist}(\overline{\theta})$. One probabilistic assignment $\ell$ may induce multiple seeds $s \in Seeds(\ell)$ from the distribution.*

**Definition 2 (Single-Path Probabilistic Dependence).** *For a data-flow fact $\langle \sigma, \gamma_{\pi,\sigma} \rangle \in f_\pi(\top)$ associated with path $\pi = [\ell_1, \ldots, \ell_n]$, its single-path probabilistic dependence is $\gamma_{\pi,\sigma} = [s_1, \ldots, s_m]$, which consists of a sequence of probability seeds (Definition 1) based on all the probabilistic assignments on $\pi$ ($m < n$). The probability of $\sigma$ for $\pi$ is $\rho_{\pi,\sigma} = Pr(\gamma_{\pi,\sigma}) = Pr(s_1) * Pr(s_2) * \cdots * Pr(s_m)$.*

**Definition 3 (All-Path Probabilistic Dependence).** *For a data-flow fact $\langle \sigma, \gamma_\sigma \rangle \in D_\ell$ at the beginning of $\ell$, its all-path probabilistic dependence $\gamma_\sigma = \{\gamma_{\pi,\sigma} \mid \pi \in paths(\ell)\}$ consists of the dependence information for every single path $\pi$ reaching $\ell$, with $\sigma$'s probability being $\rho_\sigma = Pr(\gamma_\sigma) = \Sigma_{\pi \in paths(\ell)}\ \rho_{\pi,\sigma}$.*

Let us look at the example in Figure 5 to illustrate how Icpp incrementally computes the posterior probability distributions at $\ell_6$ once the prior probability distribution at a probabilistic assignment is changed. *Pre-inference* is first performed to generate the probabilistic dependences for all the data-flow facts during the on-the-fly data-flow analysis. Based on the dependence information, *sparse incremental update* is performed to recalculate the posterior probability distributions of the existing data-flow facts at $\ell_6$ affected by the change made. Finally, we propagate the new data-flow facts introduced by the change across the entire program in a sparse manner via *sparse incremental propagation*.

***Pre-inference.*** For the program given in Figure 5(a), the data-flow facts obtained by pre-inference are listed in Figure 5(b).

To start with, the probabilistic assignment $\ell_1$ based on the Bernoulli distribution assigns a random value 0 or 1 to variable $a$ with each value's probability being 1/2. As shown, $D_{\ell_2}$ therefore contains the two data-flow facts, where the probabilistic dependence of each state is its corresponding probability seed generated from $\ell_1$ (e.g., $(a = 0)$ is annotated with its seed $[\ell_1 : a = 0]$).

The probabilistic assignment at $\ell_2$ gives variable $b$ a random value, 0 or 1, based on a discrete uniform distribution. By combining with the two values of variable $a$, we obtain the four data-flow facts in $D_{\ell 3}$ to represent the four possible states for $a$ and $b$ with the probability of each state being 1/4. The corresponding probabilistic dependence of each state (e.g., $(a = 0, b = 0)$) is a sequence of probability seeds (e.g., $[\ell_1 : a = 0, \ell_2 : b = 0]$), which are used to compute its corresponding probability (e.g., $\rho_{(a=0,b=0)} = Pr(\ell_1 : a = 0) * Pr(\ell_2 : b = 0) = 1/4$).

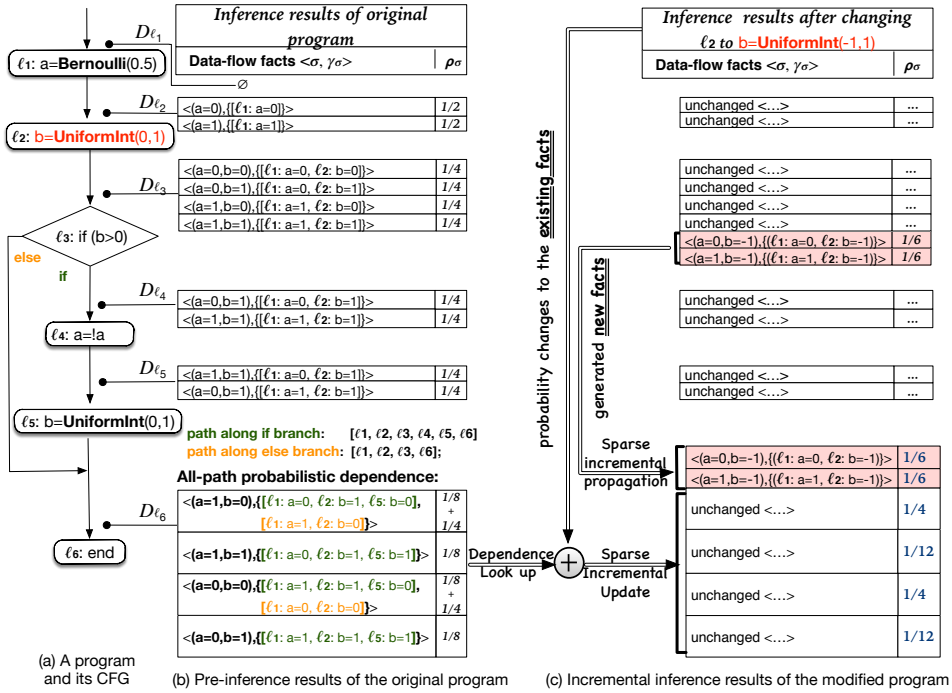


**Fig. 5.** A motivating example illustrating how ICPP works in response to the change at $\ell_2$ from `UniformInt`(0,1) to `UniformInt`$(-1, 1)$ (as highlighted in red).

There are two branches at $\ell_3$ when propagating $D_{\ell_3}$ forward. Only two data-flow facts whose states satisfy condition $b > 0$ are propagated to the `if` branch as illustrated in $D_{\ell_4}$ while the other two are propagated to the `else` branch. After analyzing $\ell_4$, $a$'s value in each data-flow fact of $D_{\ell_4}$ is flipped, while $b$'s value stays the same. Note that the probabilistic dependence recorded in each data-flow fact remains unchanged, as indicated in $D_{\ell_5}$, because $\ell_4$ is a deterministic statement, which does not affect any probabilistic dependence in any way.

After analyzing $\ell_5$, $D_{\ell_6}$ contains six data-flow facts at the join point before $\ell_6$ (the end of the program). The four data-flow facts highlighted in green are generated after analyzing $\ell_5$ in the `if` branch and the two data-flow facts in orange are propagated from the `else` branch. Let $\pi_{\text{if}} = [\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6]$ and $\pi_{\text{else}} = [\ell_1, \ell_2, \ell_3, \ell_6]$ as shown in Figure 5(b). The data-flow facts whose states are the same are merged by computing their all-path probabilistic dependence (Definition 3). Therefore, $\langle (a = 0, b = 0), \gamma_{\pi_{\text{if}},(a=0,b=0)} \rangle$ from the `if` branch and $\langle (a = 0, b = 0), \gamma_{\pi_{\text{else}},(a=0,b=0)} \rangle$ from the `else` branch are merged into $\langle (a = 0, b = 0), \{\gamma_{\pi_{\text{if}},(a=0,b=0)}, \gamma_{\pi_{\text{else}},(a=0,b=0)}\} \rangle$, where

$$\begin{aligned} \gamma_{\pi_{\text{if}},(a=0,b=0)} &= [\ell_1 : a = 1, \ell_2 : b = 1, \ell_5 : b = 0] \\ \gamma_{\pi_{\text{else}},(a=0,b=0)} &= [\ell_1 : a = 0, \ell_2 : b = 0] \end{aligned} \tag{2}$$

Likewise, the two data-flow facts with the same state $(a = 1, b = 0)$ are also merged. Finally, we calculate the joint posterior probability $\rho_\sigma$ for each data-flow fact reaching $\ell_6$ based on its probabilistic dependence as shown in Figure 5(b).

***Sparse Incremental Update.*** Here, our incremental analysis is concerned with updating the posterior probabilities of the existing data-flow facts in $D_{\ell_6}$, which are affected by the changes made to the prior probability distributions discovered by the computed probabilistic dependences. ICPP does not reanalyze the program to recompute any of the existing data-flow facts $\langle \sigma, \gamma_\sigma \rangle \in D_{\ell_6}$. Instead, it just recalculates its posterior probability $\rho_\sigma$. For example, the probabilistic assignment $\ell_2$, changed from $b = \texttt{UniformInt}(0, 1)$ to $b = \texttt{UniformInt}(-1, 1)$, causes the prior probabilities of the two probability seeds to change from $Pr^{old}(\ell_2 : b = 0) = Pr^{old}(\ell_2 : b = 1) = 1/2$ to $Pr^{new}(\ell_2 : b = 0) = Pr^{new}(\ell_2 : b = 1) = 1/3$. In this motivating example, we are interested in the effects of the change on the posterior probabilities at $\ell_6$. As shown in Figure 5(b), $D_{\ell_6}$ contains four data-flow facts that are computed before the change is made. Therefore, their probabilities need to be updated. Consider first $\langle (a = 0, b = 0), \{\gamma_{\pi_{\text{if}},(a=0,b=0)}, \gamma_{\pi_{\text{else}},(a=0,b=0)}\}\rangle$, where $\gamma_{\pi_{\text{if}},(a=0,b=0)}$ and $\gamma_{\pi_{\text{else}},(a=0,b=0)}$ are given in (2). The following equation recalculates the posterior probability of its corresponding state $(a = 0, b = 0)$, whose probabilistic dependence contains the two probability seeds, $\ell_2 : b = 0$ and $\ell_2 : b = 1$:

$$\begin{aligned}\rho^{new}_{(a=0,b=0)} &= \frac{Pr^{old}(\gamma_{\pi_{\text{if}},(a=0,b=0)}) * Pr^{new}(\ell_2 : b = 1)}{Pr^{old}(\ell_2 : b = 1)} \\ &\quad + \frac{Pr^{old}(\gamma_{\pi_{\text{else}},(a=0,b=0)}) * Pr^{new}(\ell_2 : b = 0)}{Pr^{old}(\ell_2 : b = 0)} \\ &= \frac{1/8 * 1/3}{1/2} + \frac{1/4 * 1/3}{1/2} = 1/4\end{aligned}$$

Likewise, the probabilities of the other three data-flow facts in $D_{\ell_6}$ are updated as $\rho^{new}_{(a=1,b=0)} = 1/4$, $\rho^{new}_{(a=1,b=1)} = 1/12$ and $\rho^{new}_{(a=0,b=1)} = 1/12$. These updated posterior probabilities are reflected in the bottom of Figure 5(c).

Updating existing data-flow facts incrementally this way is lightweight. As we are interested in the effects of a change on $\ell_6$ in our motivating example, the posterior probabilities for the data-flow facts in $D_{\ell_6}$ are recalculated directly. All the other data-flow facts from $D_{\ell_1}$ to $D_{\ell_5}$ remain untouched, without requiring any expensive data-flow analysis that computes and propagates data-flow facts (probabilistic dependences) along the program's control-flow.

***Sparse Incremental Propagation.*** The change made to the prior probability distribution at $\ell_2$ also introduces a new probability seed $[\ell_2 : b = -1]$ with its probability $Pr(\ell_2 : b = -1) = 1/3$, as illustrated in Figure 5(c). During the sparse incremental propagation, the two new data-flow facts, $\langle (a = 0, b = -1), \{[\ell_1 : a = 0, \ell_2 : b = -1]\}\rangle$ and $\langle (a = 1, b = -1), \{[\ell_1 : a = 1, \ell_2 : b = -1]\}\rangle$, are generated and appended to $D_{\ell_3}$. In general, the new data-flow facts generated this way are

propagated sparsely along the control flow in the program, without causing the existing data-flow facts to be modified. Finally, we obtain the updated posterior joint distributions at $\ell_6$ by combining the results of both existing and new data-flow facts incrementally computed for $\ell_6$, as shown in Figure 5(c).

## 4 ICPP: Incremental Analysis for Probabilistic Programs

In this section, we describe our pre-inference and incremental inference, which are conducted in response to changes made to probabilistic knowledge at probabilistic assignments and/or observe statements.

### 4.1 Pre-inference

The probabilistic dependence analysis during pre-inference forms the basis for ICPP. It takes a probabilistic program as input and produces as output the data-flow facts with a probabilistic dependence $\gamma_\sigma$ over each state $\sigma$ of the program. Figure 6 gives our algorithm, which introduces the transfer functions for analyzing each type of statements in Figure 2 by computing the data-flow facts in a forward traversal of the CFG of the program being analyzed.

**4.1.1 Notations.** We adopt some notations from [8]. For a state $\sigma$, $\sigma(x)$ denotes the value of variable $x$ in $\sigma$. Likewise, the notation $\sigma(\mathcal{E})$ evaluates the value of expression $\mathcal{E}$ in $\sigma$. $\sigma[x \leftarrow \sigma(x)]$ represents the state obtained by updating the value of $x$ in $\sigma$, with the values of all the other variables in $\sigma$ remaining unchanged. The function $ite(b, x, y)$ evaluates to $x$ if $b = true$ and $y$ if $b = false$.

Given a statement $\ell$, $\Omega_\ell$ is used to denote all the states recorded in the data-flow facts of $D_\ell$. For the purposes of explaining our algorithm cleanly, $D_\ell$ is represented by a lambda function $\lambda_\sigma.expr$, where each state $\sigma \in \Omega_\ell$ is bounded in expression $expr$, which represents the all-path probabilistic dependence of $\sigma$. By default, we define $\top = \lambda_\sigma.\emptyset$. For $(\sigma, \gamma_\sigma) \in D_\ell$, we write $\gamma_\sigma \oplus s$ for seed collection by adding a probability seed $s$ into every single-path dependence $\gamma_{\pi,\sigma} \in \gamma_\sigma$ (where $\pi \in paths(\ell)$ ranges from all the paths reaching $\ell$ by Definition 3).

**4.1.2 Probabilistic Dependence Analysis.** Given a program $\ell \in Prog$, we call PREIN$(\top, \ell)$ (Figure 6) recursively to compute its data-flow facts.

Lines 2–3 handle a deterministic assignment $\ell : x := \mathcal{E}$, where multiple states $\sigma \in \Omega_\ell$ of the data-flow facts in $D_\ell$ may become (i.e., be merged into) the same new state $\sigma'$ after the value of $x$ is updated with a new value $\sigma(\mathcal{E})$. Consequently, the corresponding probabilistic dependences of these states $\sigma \in \Omega_\ell$ are merged together to obtain the all-path probabilistic dependence of $\sigma'$.

For each probability seed $[\ell : x = a]$ generated at a probabilistic assignment $\ell$, lines 4–5 compute new data-flow facts for all states $\sigma \in \Omega_\ell$ similarly as the case when a deterministic statement is handled, except that the all-path dependence $\gamma_\sigma$ of $\sigma$ is updated by adding the new probability seed $[\ell : x = a]$ into $\gamma_\sigma$. The

**Algorithm** PreIn($D_\ell, \ell$)
`Input`: set of data-flow facts $\langle \sigma, \gamma_\sigma \rangle \in D_\ell$ over all the states
$\sigma \in \Omega_\ell$ before analyzing $\ell$ and a statement at $\ell$
`Output`: set of data-flow facts $\langle \sigma', \gamma'_\sigma \rangle \in D'_\ell$ in the form of $\lambda_\sigma.expr$ after analyzing $\ell$
1: **switch**($\ell$)
2: **case** $x := \mathcal{E}$:
3: **return** $\lambda\sigma'. \bigcup_{\{\sigma \in \Omega_\ell \mid \sigma[x \leftarrow \sigma(\mathcal{E})] = \sigma'\}} \gamma_\sigma$ ;
4: **case** $x :=$ `Dist`($\overline{\theta}$):
5: **return** $\lambda\sigma'. \bigcup_{(\ell: x=a) \in Seeds(\ell)} ((\bigcup_{\{\sigma \in \Omega_\ell \mid \sigma[x \leftarrow a] = \sigma'\}} \gamma_\sigma) \oplus (\ell : x = a))$;
6: **case** `observe`($\mathcal{E}$):
7: **return** $\lambda\sigma.ite(\sigma(\mathcal{E}), \gamma_\sigma, \emptyset)$;
8: **case** *skip*:
9: **return** $D_\ell$;
10: **case** $\ell_1$; $\ell_2$:
11: $D_{\ell_2}$ = PreIn($D_\ell$, $\ell_1$);
12: **return** PreIn($D_{\ell_2}$, $\ell_2$);
13: **case** if $\mathcal{E}$ then $\ell_1$ else $\ell_2$:
14: $D_{\ell_1} = \lambda\sigma.ite(\sigma(\mathcal{E}), \gamma_\sigma, \emptyset)$;
15: $D_{\ell_2} = \lambda\sigma.ite(\sigma(\mathcal{E}), \emptyset, \gamma_\sigma)$;
16: **return** $\lambda\sigma'.$(PreIn($D_{\ell_1}, \ell_1$)($\sigma'$) $\bigcup$ PreIn($D_{\ell_2}, \ell_2$)($\sigma'$));
17: **case** while $\mathcal{E}$ do $\ell_1$:
18: $D_{pre} = \bot$, $D_{cur} = D_\ell$;
19: **while** KL-divergence($D_{pre}, D_{cur}$) $\neq$ *true* **do**
20: $D_{pre} = D_{cur}$;
21: $D_{cur}$ = PreIn($D_{pre}$, **if** $\mathcal{E}$ **then** $\ell_1$ **else** skip);
23: **end while**
24: **return** $\lambda\sigma'.ite(\sigma'(\mathcal{E}), \emptyset, D_{cur}(\sigma'))$;
25: **end switch**

**Fig. 6.** An algorithm for pre-inference.

set of data-flow facts obtained at a probabilistic statement $\ell$ is the union of the sets of data-flow facts computed for all its probability seeds $s \in Seeds(\ell)$ at $\ell$.

Lines 6–7 handle an observe statement `observe`($\mathcal{E}$) by simply removing the dependence information $\gamma_\sigma$ of any state $\sigma \in \Omega_\ell$ if $\sigma(\mathcal{E})$ evaluates to false. Lines 9–10 handle a sequence of two statements $\ell_1; \ell_2$ by first computing the data-flow facts for $\ell_1$ and using the resulting facts as the input to analyze $\ell_2$.

Lines 13–16 handle an if statement. Our path-sensitive analysis first splits the set of data-flow facts reaching $\ell$ into two subsets, $D_{\ell_1}$ and $D_{\ell_2}$, based on the Boolean predicate $\mathcal{E}$. Then the bodies of the `if` and `else` branches are recursively computed by applying PreIn. Finally, we return the results by merging the data-flow facts obtained from both the `if` and `else` branches.

Lines 17–24 handle a while loop by computing the results until a fixed-point is reached. We define $D_{pre}$ and $D_{cur}$ to represent the sets of previous and current data-flow facts across two consecutive iterations of the while loop. Initially, $D_{pre}$ is set as $\bot$ and $D_{cur}$ as $D_\ell$ obtained just before the while loop. PreIn is repeatedly applied to the data-flow facts in $D_{pre}$ with the statement "**if** $\mathcal{E}$ **then** $\ell_1$ **else** *skip*" until a fixed-point based on KL-divergence [17]. Due to the

non-determinism of probabilistic programs [8,10,12] (e.g., a probabilistic assignment generates a probability seed randomly during each loop iteration), finding a loop iteration under which $D_{cur} = D_{pre}$ is potentially nonterminating. Thus, we use KL-divergence to enforce the termination of a while loop. In line 19, KL-divergence$(D_{cur}, D_{pre})$ is true if the following condition holds:

$$\mid \sum_{\sigma \in \Omega_{cur}} \rho_\sigma * \ln(\frac{\rho_\sigma}{\rho'_\sigma}) \mid \; < \; threshold \tag{3}$$

where $\rho_\sigma$ is the probability of $\sigma \in \Omega_{cur}$ calculated based on $\langle \sigma, \gamma_\sigma \rangle \in D_{cur}$, $\rho'_\sigma$ is the probability calculated based on $\langle \sigma, \gamma'_\sigma \rangle \in D_{pre}$, and *threshold* is a user-determined parameter (set to 0.01 in our experiments). Note that a probability seed $s$ may appear multiple times in a single-path dependence when a fixed-point is reached. For example, $\gamma_{\pi,\sigma} = [s, s, \ldots, s]$ if the path $\pi$ contains some loops.

```
ℓ1: b=0;
ℓ2: while(!b) do {
ℓ3:   b=Bernoulli(0.5)
ℓ4:  }
```

**Fig. 7.** An OneCoin example.

*Example 1.* Let us use a simple OneCoin program in Figure 7 to explain KL-divergence in a while loop. At the $k$-th iteration, there are two states, $(b = 0)$ and $(b = 1)$, with their all-path probabilistic dependences being $\gamma_{(b=0)} = \{[\ell_3 : b = 0, \ell_3 : b = 0; \cdots]\}$ and $\gamma_{(b=1)} = \{[\ell_3 : b = 1], [\ell_3 : b = 0, \ell_3 : b = 1], [\ell_3 : b = 0, \ell_3 : b = 0, \ell_3 : b = 1], \cdots\}$ immediately after $\ell_3$. Their corresponding probabilities are $\rho_{(b=0)} = (0.5)^k$ and $\rho_{(b=1)} = 0.5 + ... + (0.5)^k$. Thus, the KL-divergence between iterations $k$ and $k-1$ is computed as follows:

$$(0.5)^k \times \ln(\frac{(0.5)^k}{(0.5)^{k-1}}) + (0.5 + ... + (0.5)^k) \times \ln(\frac{0.5 + ... + (0.5)^k}{0.5 + ... + (0.5)^{k-1}})$$

□

Let us revisit the example in Figure 5 to go through our pre-inference algorithm in Figure 6. Given the program $\ell_1; \ell_2; \ell_3; \ell_6$ in Figure 5(a), we see how calling PreIn$(\top, \ell_1; \ell_2; \ell_3; \ell_6)$ yields the data-flow facts obtained in Figure 5(b).

*Example 2.* The sequence $\ell_1; \ell_2; \ell_3; \ell_6$ is analyzed in order, starting from $\ell_1 : a =$ `Bernoulli(0.5)` (lines 10–12). $\ell_1$ generates two probability seeds, $\ell_1 : a = 0$ and $\ell_1 : a = 1$ (lines 4–5). Thus, we obtain two states, $(a = 0)$ and $(a = 1)$, which are recorded in $\Omega_{\ell_2}$. Their probability seeds are added to their probabilistic dependences, resulting in $\gamma_{(a=0)} = \{[\ell_1 : a = 0]\}$ and $\gamma_{(a=1)} = \{[\ell_1 : a = 1]\}$. As a result, $D_{\ell_2}$ contains the two data-flow facts, as shown in Figure 5(b).

When analyzing $\ell_2 : b =$ `UniformInt(0, 1)` (lines 4–5), we obtain also two probability seeds, $\ell_2 : b = 0$ and $\ell_2 : b = 1$. By combining each seed with each of

the two states in $\Omega_{\ell_2} = \{(a = 0), (a = 1)\}$, we obtain the four states in $D_{\ell_3}$, as shown in Figure 5(b), with the probabilistic dependence $\gamma_\sigma$ of each state $\sigma \in \Omega_{\ell_3}$ containing an additional probability seed of either $\ell_2 : b = 0$ or $\ell_2 : b = 1$. As a result, $D_{\ell_3}$ contains the four data-flow facts, as shown in Figure 5(b).

When analyzing $\ell_3$ : **if** $(b > 0)$ **then** $\ell_4; \ell_5$ **else** *skip* (lines 13–16), we split $D_{\ell_3}$ into $D_{\ell_4;\ell_5}$ and $D_{skip}$ according to the condition $b > 0$, where $D_{\ell_4;\ell_5}$ is propagated into the `if` branch and and $D_{\text{skip}}$ into the `else` branch. Then we continue to apply PreIn to compute the data-flow facts for $\ell_4; \ell_5$ (lines 10–12) and *skip* (lines 8–9). Finally, we merge the data-flow facts flowing out of the two branches at the beginning of $\ell_6$ to obtain $D_{\ell_6}$ (line 16). □

### 4.2 Incremental Inference

Based on the computed probabilistic dependence information, our incremental analysis handles two type of changes made to a probabilistic program, i.e, prior distribution changes at a probabilistic assignment (Section 4.2.1) and condition changes at an observe statement (Section 4.2.2). Icpp aims to recalculate the posterior probability $\rho_\sigma$ for each data-flow fact $\langle \sigma, \gamma_\sigma \rangle \in D_{\ell_{end}}$ at $\ell_{end}$ (the end of a program) according to the computed probabilistic dependence $\gamma_\sigma$, in response to the changes made to probabilistic knowledge in the program.

Without loss of generality, we restrict ourselves to a change made to one single statement at a time. Our incremental inference generalizes straightforwardly to the changes made simultaneously to multiple statements.

**4.2.1 Handling Changes Made at Probabilistic Assignments.** For a probabilistic assignment $x = \texttt{Dist}(\overline{\theta})$, Icpp focuses on a change made to the prior distribution $\texttt{Dist}(\overline{\theta})$, which is defined over a measurable sample space with a probability measure. Thus, a change can be a modification of the sample space or the probability measure. For example, if $x = \texttt{Bernoulli}(0.5)$ is modified to $x = \texttt{Bernoulli}(0.6)$, the sample space, $\{0, 1\}$, remains the same, but the probability measure is adjusted, with the probability of $x = 1$ changed from 0.5 and 0.6. However, modifying $x = \texttt{UniformInt}(0, 1)$ into $x = \texttt{UniformInt}(-1, 1)$ will change both its sample space and probability measure. Similarly, modifying a distribution model from `Dist` to `Dist′` also affects both.

Modifying a probability measure changes the posterior probabilities of existing data-flow facts computed by pre-inference. Modifying a sample space generates new probability seeds, and consequently, introduces new data-flow facts.

For a change made at a probabilistic assignment, the algorithm in Figure 8 updates the posterior probabilities of the existing data-flow facts affected via IncUpdate and propagates the newly introduced data-flow facts via IncProp.

***Sparse Incremental Update.*** According to our algorithm in Figure 8, $S^{com} = Seeds(\ell_{old}) \cap Seeds(\ell_{new})$ is the set of probability seeds that exist in both the original and modified programs. $D_{\ell_{end}}$ is the set of data-flow facts that reach $\ell_{end}$ computed before the change. IncUpdate$(D_{\ell_{end}}, S^{com})$ can instantly recalculate

HandleProbAssign($\ell_{old} : x = \texttt{Dist}(\overline{\theta})$, $\ell_{new} : x = \texttt{Dist}'(\overline{\theta}')$)
1: $S^{com} = Seeds(\ell_{old}) \cap Seeds(\ell_{new})$;
2: Let $D_{\ell_{end}}$ be the set of existing data-flow facts before the end of the program;
3: $\Psi_1$=IncUpdate($D_{\ell_{end}}, S^{com}$); // handling the existing data-flow facts
4: $D^{\Delta} = \text{PreIn}(D_{\ell_{old}}, \ell_{new}) \setminus \text{PreIn}(D_{\ell_{old}}, \ell_{old})$;
5: $\Psi_2$=IncProp($D^{\Delta}$); // handling the new data-flow facts
6: **return** $\Psi_1 \cup \Psi_2$;

**Fig. 8.** An algorithm for performing incremental analysis due to a change made from $\ell_{old} : x = \texttt{Dist}(\overline{\theta})$ to $\ell_{new} : x = \texttt{Dist}'(\overline{\theta}')$ at a probabilistic assignment. $\Psi_1$ and $\Psi_2$ are the posterior distributions obtained in analyzing the existing and new data-flow facts.

the posterior probability distributions for the states in $D_{\ell_{end}}$ based on the all-path probabilistic dependence $\gamma_\sigma$ computed by pre-inference for each data-flow fact in $D_{\ell_{end}}$, without a need for performing any data-flow analysis.

The new posterior probability distributions for $D_{\ell_{end}}$ are obtained directly:

$$\text{IncUpdate}(D_{\ell_{end}}, S^{com}) = \{\langle \sigma, \rho_\sigma^{new} \rangle \mid \langle \sigma, \gamma_\sigma \rangle \in D_{\ell_{end}}\} \tag{4}$$

with the new posterior probability $\rho_\sigma^{new} = Cal(\gamma_\sigma, S^{com})$ being obtained as:

$$Cal(\gamma_\sigma, S^{com}) = \sum_{(\gamma_{\pi,\sigma}, S) \in \mathsf{Affected}_\sigma} Pr^{old}(\gamma_{\pi,\sigma}) \times \prod_{s \in S} \frac{Pr^{new}(s)}{Pr^{old}(s)} + \sum_{\gamma_{\pi,\sigma} \in \mathsf{NotAffected}_\sigma} Pr^{old}(\gamma_{\pi,\sigma}) \tag{5}$$

where $\mathsf{Affected}_\sigma = \{(\gamma_{\pi,\sigma}, S) \mid \gamma_{\pi,\sigma} \in \gamma_\sigma, S = \{s \mid s \in \gamma_{\pi,\sigma} \wedge s \in S^{com}\}\}$, which consists of a set of pairs with each $(\gamma_{\pi,\sigma}, S)$ representing the fact that the single-path dependence $\gamma_{\pi,\sigma} \in \gamma_\sigma$ is affected by some seeds in $S$ whose probabilities are changed, on the path $\pi$ containing $\ell_{old}$. Note that $S$ is a multiset as it may contain multiple instances of a seed $s$ from $\gamma_{\pi,\sigma}$ due to loops on $\pi$.

We also define $\mathsf{NotAffected}_\sigma = \{\gamma_{\pi,\sigma} \in \gamma_\sigma \mid \forall\ s \in \gamma_{\pi,\sigma} : s \notin Seeds(\ell_{old})\}$. This contains the single-path dependences such that each $\gamma_{\pi,\sigma}$ is not affected by any seed in $Seeds(\ell_{old})$ generated by the old statement $\ell_{old}$, i.e., $\ell_{old}$ is not on $\pi$.

The probability of $\gamma_{\pi,\sigma}$ is set to 0 if $\gamma_{\pi,\sigma}$ contains any seed $s \in (Seeds(\ell_{old}) \setminus Seeds^{com}_{\ell})$, which will be removed from the modified program.

Finally, $Pr^{old}(s)$ and $Pr^{new}(s)$ represent the probabilities of seed $s$ in the original and modified programs, respectively.

*Example 3.* Let us revisit the example in Figure 5(c) to explain our incremental update for an existing data-flow fact $\langle (a = 0, b = 0), \gamma_{(a=0,b=0)} \rangle \in D_{\ell_6}$. Recall that $\gamma_{(a=0,b=0)} = \{\gamma_{\pi_{\text{if}},(a=0,b=0)}, \gamma_{\pi_{\text{else}},(a=0,b=0)}\}$, where $\gamma_{\pi_{\text{if}},(a=0,b=0)}$ and $\gamma_{\pi_{\text{else}},(a=0,b=0)}$ are from (2). Given the change from $\ell_{old} : b = \texttt{UniformInt}(0, 1)$ to $\ell_{new} : b = \texttt{UniformInt}(-1, 1)$, we have $S^{com} = Seeds(\ell_{old}) = \{\ell_2 : b = 0, \ell_2 : b = 1\}$. Thus, $\mathsf{Affected}_{(a=0,b=0)} = \{(\gamma_{\pi_{\text{if}},(a=0,b=0)}, \{[\ell_2 : b = 1]\}), (\gamma_{\pi_{\text{else}},(a=0,b=0)}, \{[\ell_2 : b = 0]\})\}$ and $\mathsf{NotAffected}_{(a=0,b=0)} = \emptyset$. Based on (5), we obtain:

$$\begin{aligned} Cal(\gamma_{(a=0,b=0)}, S_{com}) &= \frac{Pr^{old}(\gamma_{\pi_{\text{if}},(a=0,b=0)}) * Pr^{new}(\ell_2 : b = 1)}{Pr^{old}(\ell_2 : b = 1)} + \frac{Pr^{old}(\gamma_{\pi_{\text{else}},(a=0,b=0)}) * Pr^{new}(\ell_2 : b = 0)}{Pr^{old}(\ell_2 : b = 0)} \\ &= \frac{1/8 * 1/3}{1/2} + \frac{1/4 * 1/3}{1/2} = 1/4 \end{aligned}$$

□

***Sparse Incremental Propagation.*** According to Figure 8, we first collect $D^{\Delta}$, the set of new data-flow facts introduced by comparing the data-flow facts obtained after analyzing $\ell_{new}$ and $\ell_{old}$. Then we make use of $\textsc{IncProp}(D^{\Delta})$ to perform incremental propagation by calling $\textsc{PreIn}(D^{\Delta}, L)$, where $L$ is the set of statements $L$ reachable from $\ell_{old}$ on the CFG of the program being analyzed:

$$\textsc{IncProp}(D^{\Delta}) = \{\langle \sigma, Pr(\gamma_\sigma)\rangle \mid \langle \sigma, \gamma_\sigma\rangle \in D^{\Delta}_{\ell_{end}} = \textsc{PreIn}(D^{\Delta}, L)\} \quad (6)$$

*Example 4.* Let us still consider the example in Figure 5(c). For the change made from $b = \texttt{UniformInt}(0,1)$ to $b = \texttt{UniformInt}(-1,1)$ at $\ell_2$, we first collect the two new data-flow facts introduced by the change at $\ell_2$: $D^{\Delta} = \{\langle (a=0, b=-1), \{[\ell_1 : a=0, \ell_2 : b=-1]\}\rangle, \langle (a=1, b=-1), \{[\ell_1 : a=1, \ell_2 : b=-1]\}\rangle\}$. In this case, $L = \{\ell_3, \ell_6\}$. Following (6), we then call $\textsc{PreIn}(D^{\Delta}, \{\ell_3, \ell_6\})$ to obtain the two new data-flow facts in $D^{\Delta}_{\ell_6}$ highlighted in red at the end of the program by incrementally propagating the two new data-flow facts in $D^{\Delta}$ across the CFG of the program without affecting any of the existing data-flow facts. □

**4.2.2 Handling Changes Made at Observe Statements.** In our data-flow analysis, an observe statement $\ell : \texttt{observe}(\mathcal{E})$ filters out any data-flow fact $\langle \sigma, \gamma_\sigma\rangle$, whose state $\sigma$ violates the condition $\mathcal{E}$ by blocking the propagation of $\langle \sigma, \gamma_\sigma\rangle$ after analyzing $\ell$. All the others satisfying $\mathcal{E}$ remain unchanged.

For a modification of a probabilistic assignment $\ell$, we find any existing data-flow fact $\langle \sigma, \gamma_\sigma\rangle \in D_{\ell_{end}}$ affected by the change and update its probability based on the new seeds generated at $\ell$. However, for a modification of an observe statement, we will need to find any $\langle \sigma, \gamma_\sigma\rangle \in D_{\ell_{end}}$ affected by the change based on the dependence information from one or more probabilistic assignments. This is because the value $\mathcal{E}$ in $\texttt{observe}(\mathcal{E})$ may be affected by multiple probabilistic assignments. For example, $\texttt{observe}(a||b)$ contains $a||b$, where $a$ and $b$ may be defined by two different Bernoulli assignments in the program.

$\textsc{HandleObserve}(\ell_{old} : \texttt{observe}(\mathcal{E}),\ \ell_{new} : \texttt{observe}(\mathcal{E}'))$
1: $D^{diff} = \textsc{PreIn}(D_{\ell_{old}}, \ell_{old}) \setminus \textsc{PreIn}(D_{\ell_{old}}, \ell_{new})$;
2: $\Gamma^{diff} = \bigcup_{\langle \sigma, \gamma_\sigma\rangle \in D^{diff}} \gamma_\sigma$;
3: Let $D_{\ell_{end}}$ be the existing data-flow facts before the end of the program;
4: $\Psi_1 = \textsc{IncUpdate}^{\sharp}(D_{\ell_{end}}, \Gamma^{diff})$; // handling the existing data-flow facts
5: $D^{\Delta} = \textsc{PreIn}(D_{\ell_{old}}, \ell_{new}) \setminus \textsc{PreIn}(D_{\ell_{old}}, \ell_{old})$;
6: $\Psi_2 = \textsc{IncProp}(D^{\Delta})$; // handling the new data-flow facts
7: **return** $\Psi_1 \cup \Psi_2$;

**Fig. 9.** An algorithm for performing incremental analysis due to a change made from $\texttt{observe}(\mathcal{E})$ to $\texttt{observe}(\mathcal{E}')$. $\Psi_1$ and $\Psi_2$ are the posterior distributions obtained in analyzing the existing and new data-flow facts.

Our algorithm given in Figure 9 for handling an observe statement is the same as the one for handling a probabilistic assignment given in Figure 8, except that

IncUpdate in Figure 8 is replaced by IncUpdate$^\sharp$ in order to deal with existing data-flow facts in $D_{\ell_{end}}$. Unlike IncUpdate, which uses the seeds in $S^{com}$ collected from only one probabilistic assignment $\ell$ (modified), IncUpdate$^\sharp$ uses $\Gamma^{diff}$, which contains a set of single-path dependences possibly from multiple probabilistic assignments based on the data-flow facts in $D^{diff}$ that exist in the original program but not the modified program (lines 1–2).

At line 5, IncProp is reused based on (6) to propagate the new data-flow facts that were filtered out by the original observe statement but become valid after the change. At line 6, we obtain the new posterior probability distributions at $\ell_{end}$ by combining the results from incremental update and propagation.

To update the posterior probability distributions of the states in $D_{\ell_{end}}$, we first find a set of affected single-path probabilistic dependences: $\mathsf{Affected}_\sigma = \{\gamma_{\pi,\sigma} \in \gamma_\sigma \mid \exists\ \gamma \in \Gamma^{diff} : \gamma \subseteq \gamma_{\pi,\sigma}\}$ for each fact $\langle \sigma, \gamma_\sigma \rangle \in D_{\ell_{end}}$. These affected dependences are no longer existent in the modified program according to $\Gamma^{diff}$ due to the change made at $\ell$, Thus, they are excluded with their old probabilities set to 0. We only recalculate the posterior probabilities based on $\mathsf{NotAffected}_\sigma = \{\gamma_{\pi,\sigma} \mid \gamma_{\pi,\sigma} \in (\gamma_\sigma \setminus \mathsf{Affected}_\sigma)\}$, which contains the single-path dependences that are not affected by the change. Therefore, the new posterior distributions for the states in $D_{\ell_{end}}$ are computed as:

$$\text{IncUpdate}^\sharp(D_{\ell_{end}}, \Gamma^{diff}) = \{\langle \sigma, \rho_\sigma^{new} \rangle | \langle \sigma, \gamma_\sigma \rangle \in D_{\ell_{end}}\} \tag{7}$$

with the new posterior probability $\rho_\sigma^{new} = Cal^\sharp(\gamma_\sigma, \Gamma^{diff})$ being obtained by:

$$Cal^\sharp(\gamma_\sigma, \Gamma^{diff}) = \sum_{\gamma_{\pi,\sigma} \in \mathsf{Affected}_\sigma} Pr^{old}(\gamma_{\pi,\sigma}) \times 0 + \sum_{\gamma_{\pi,\sigma} \in \mathsf{NotAffected}_\sigma} Pr^{old}(\gamma_{\pi,\sigma}) \tag{8}$$

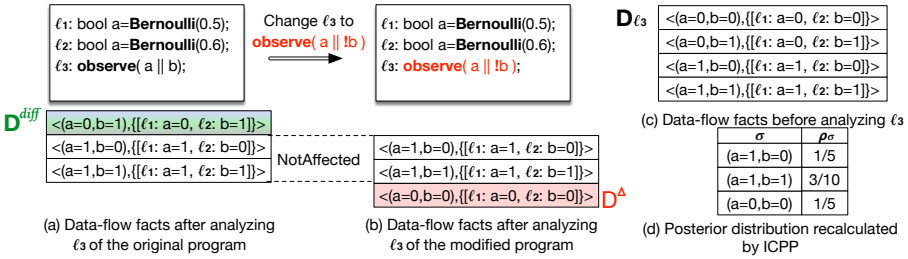


**Fig. 10.** An example for incremental analysis of an observe statement.

*Example 5.* Figure 10 illustrates our incremental analysis for handing a change at an observe statement. In Figure 10(a), its top part shows a small program containing an observe statement, `observe`$(a||b)$, at $\ell_3$. In Figure 10(b), its top part shows the same program with the observe statement changed to `observe`$(a||!b)$. Figure 10(c) gives the data-flow facts in $D_{\ell_3}$ obtained just before either observe statement. In Figures 10(a) and (b), their bottom parts give the data-flow facts after analyzing their observe statements in terms of the data-flow facts in $D_{\ell_3}$.

We then obtain the single data-flow fact in $D^{diff}$ blocked by the new observe statement as highlighted in green (Figure 10(a)). Thus, we have $\Gamma^{diff} = \{[\ell_1 : a =$

$0, \ell_2 : b = 1]\}$, $\mathsf{Affected}_\sigma = \{[\ell_1 : a = 0, \ell_2 : b = 1]\}$, and $\mathsf{NotAffected}_\sigma = \{[\ell_1 : a = 1, \ell_2 : b = 0], [\ell_1 : a = 1, \ell_2 : b = 1]\}$. Based on (8), we recalculate the posterior probability of each state as $\rho^{new}_{(a=0,b=1)} = Pr([\ell_1 : a = 0, \ell_2 : b = 1]) \times 0 = 0$, $\rho^{new}_{(a=1,b=0)} = Pr([\ell_1 : a = 1, \ell_2 : b = 0]) = 1/5$, and $\rho^{new}_{(a=1,b=1)} = Pr([\ell_1 : a = 1, \ell_2 : b = 1]) = 3/10$.

The data-flow fact in $D^\Delta$ as highlighted in red in Figure 10(b) is a new one introduced by the change. Finally, we combine the computed probabilities of the existing and new data-flow facts to obtain the posterior probability distributions given in Figure 10(d). Note that $1/5 + 3/10 + 1/5 \neq 1$ due to the observe statement. Thus, after having computed the posterior probabilities as desired, we normalize these probabilities as $2/7, 3/7$ and $2/7$, respectively. □

### 4.3 Precision

**Theorem 1.** Icpp *achieves the same precision as DFI [8] (which analyzes a program from scratch) in terms of answering posterior probability distributions under the changes made to the probabilistic knowledge of a probabilistic program.*

*Proof. The pre-inference of* Icpp *(Figure 6) captures the all-path dependence (Definition 3) of each data-flow fact in order to allow the posterior probability distributions to be updated during the incremental analysis. Every loop in the program is handled by approximating a KL-divergence between its two consecutive loop iterations. A continuous prior distribution is approximated by a discrete distribution over a finite set, following [8,21].*

*Based on the dependence information, our incremental sparse update recalculates the posterior probability of any existing data-flow fact affected by any change to a probabilistic assignment based on (4) or an observe statement based on (7) while keeping the probabilities of unaffected dependences unchanged. Our incremental sparse propagation computes and propagates any new data-flow fact introduced by the changes along the CFG based on (6). Following the algorithms in Figures 8 and 9, we can obtain the same posterior probability distributions as the program is reanalyzed entirely by DFI (or our pre-inference).* □

## 5 Evaluation

Our objective is to demonstrate that Icpp is effective in inferring the posterior distributions incrementally in response to small yet frequent changes made to a probabilistic program. Icpp is an order of magnitude faster than DFI [8], a state-of-the-art data-flow-based inference. Our experiment is conducted on a 2.70 GHz `Intel` Core i5 processor system with 8 GB RAM running `macOS`.10.12.4.

We have implemented Icpp in Soot [34], a Java analysis framework. We choose Figaro [27] as our probabilistic language, which is based on Scala and can be translated into the .class format for our analysis in Soot. Following DFI [8], we use the ADD library [30] to store our data-flow facts, i.e., probabilistic dependences over states, with each single-path dependence naturally represented by an ADD. Updating data-flow facts affected by changes to probabilistic assignments and observe statements is done by the graph operations in ADD.

**Table 1.** Analysis times of DFI and ICPP (seconds).

| Program | Analyzing original program | Analyzing 10 changes | | Speedup |
|---|---|---|---|---|
| | DFI/PREIN$_{\text{ICPP}}$ | DFI | ICPP | |
| `BurglarAlarm` | 1.27 / 1.38 | 12.60 | 0.65 | 19 |
| `NoisyOr` | 1.85 / 2.22 | 19.01 | 1.70 | 11 |
| `Grass` | 1.91 / 2.31 | 21.85 | 1.71 | 13 |
| `Grade` | 1.31 / 1.57 | 12.78 | 0.93 | 13 |
| `Loopy` | 1.53 / 1.72 | 15.15 | 1.47 | 10 |
| `MotWhile` | 1.46 / 1.68 | 15.51 | 1.34 | 13 |

To cover different change scenarios, we have selected a set of 6 probabilistic programs, with 3 using the `Bernoulli` distribution (`Grass`, `BurglarAlarm`, and `NoisyOR`) from the existing inference engine `R2` [25], 1 using a `UniformInt` (`MotWhile`) distribution by replacing `if` with `while` statement in our motivating example (Figure 5), and 2 using the `Bernoulli` distribution (`Grade` and `Loopy`) from [15]. `Grass`, `BurglarAlarm`, `TwoCoins` and `NoisyOR` are loop-free, `Grade` contains an observe statement in its middle, and `Loopy` and `MotWhile` contain unbounded loops (similar to Figure 5 with its `if` replaced by a `while` statement).

For each program, ten small changes are made to simulate the process of developing a probabilistic program with different versions and tuning new posterior probability distributions through each change. Note that we choose ten changes (a relatively small number) to demonstrate ICPP's effectiveness. Our incremental analysis becomes more effective than DFI [8] if more changes are added. These changes are selected to exercise as many scenarios as possible. For example, one worst-case scenario happens if a sample space is completely changed, e.g., from `UniformInt(-10,-1)` to `UniformInt (0,9)`, which requires computing all new data-flow facts without reusing any old ones. Our small modifications are made so that the probabilistic model underlying each program is not changed.

Table 1 compares ICPP with DFI [8] (which can be regarded as a special case of PREIN without recording probabilistic dependences but computing explicitly the probabilities for all the states). In Column 2, we compare the analysis times of DFI and ICPP's pre-inference in analyzing a program. Our pre-inference is slightly more costly as it must collect probabilistic dependences for all the states. In Columns 3 and 4, we compare DFI and ICPP in terms of the total analysis time spent for the 10 changes made in a program. ICPP is an order of magnitude faster than DFI. For each program, DFI must reanalyze it from scratch after each change. In contrast, ICPP performs incremental analysis for the program based on the probabilistic dependences computed during its pre-inference.

For the 5 programs with the `Bernoulli` distribution, which does not introduce new data-flow facts when prior probabilities are changed, their posterior distributions can be directly recomputed by INCUPDATE and INCUPDATE$^\sharp$, without using INCPROP. ICPP updates existing results instantaneously with negligible overheads. For `MotWhile` with a `uniform` distribution, ICPP also achieves a significant performance improvement over DFI by 13x.

Figure 11 shows that ICPP is much faster than DFI for three representative programs, `Grass`, `Loopy` and `MotWhile`. For `Grass` with the `Bernoulli` distribution, ICPP has negligible overheads for all the 10 changes made (Figure 11(a)). For `Loopy` with a `uniform` distribution containing an unbounded loop, ICPP spends relatively more time for the fourth and sixth changes (Figure 11(b)) due to the modifications of a probabilistic assignment in its unbounded loop, affecting the probability of a single-path dependence when the loop is analyzed until KL-divergence. For `MotWhile` with a `UniformInt` distribution, ICPP takes relatively more time in handling the fourth change (Figure 11(c)), because the change is made to a probabilistic assignment with a `Uniform` distribution, causing new data-flow facts to be propagated repeatedly inside a loop.

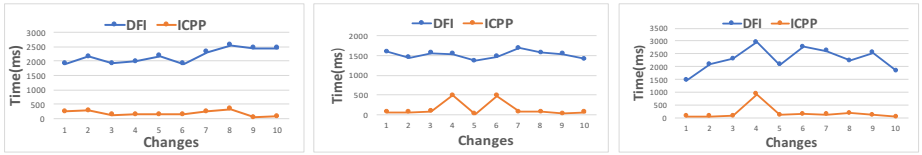


**Fig. 11.** Analysis times of ICPP and DFI over ten changes made in a program.

## 6 Related Work

In addition to the work already discussed in Section 1, we focus on the most relevant work on probabilistic inference and incremental static analysis.

***Probabilistic inference for probabilistic programs.*** The existing approaches on probabilistic inference can be classified into static and dynamic ones. Dynamic inference methods usually execute a probabilistic program a finite number of times through sampling-based Monte Carlo methods [4,7,19,25,28] and then perform inference based on the execution traces. Static methods [5,8,20,23,29] statically infer the posterior probability distributions without running the program. Sankaranarayanan et al. [29] propose a static analysis to reason about infinite-state probabilistic programs by quantifying the solution space of linear constraints over bounded floating-point domains. DFI [8] performs data-flow-based static inference that explicitly computes and maintains distributions as data-flow facts at each program point following the program's control-flow on a CFG. DFI focuses on discrete distributions and makes approximations when computing data-flow facts over continuous distributions. Recently, PSI [10] represents an exact symbolic inference for analyzing both discrete and continuous distributions for probabilistic programs with bounded loops. Based on DFI, our work enables (for the first time) efficient incremental Bayesian inference over discrete distributions with finite concrete states, in response to small yet frequent changes to probabilistic knowledge in a probabilistic program.

***Incremental static analysis for usual programs.*** The goal of incremental static analysis is to efficiently update existing analysis results without recomputing them from scratch, allowing the previously computed information to be reused. Emu [18,31] represents an incremental analysis for performing demand-driven context-sensitive pointer analysis based on Context-Free Language (CFL) reachability, which precisely recomputes points-to sets affected by the program changes. Reviser [1] is an incremental analysis technique developed as an extension to the IDE-/IFDS- based framework for efficiently updating inter-procedural data-flow analysis results. Echo [38] is an incremental analysis for data-race detection based on program dependences computed by static happens-before analysis. DiSE [26] is an incremental symbolic execution technique that uses pre-computed results from a static analysis to direct symbolic execution for exploring only the parts of a program affected by the changes. Unlike previous incremental analysis techniques for imperative programs emphasizing on code changes, i.e., statement addition and deletion, ICPP focuses on changes made to probabilistic knowledge, the key feature in probabilistic programming.

## 7 Conclusion and Future work

In this paper, we present ICPP, a new data-flow based incremental analysis for analyzing probabilistic programs. ICPP captures the correlation relation between prior and posterior probability distributions through a probabilistic dependence analysis. The resulting analysis significantly improves the efficiency of data-flow based inference by incrementally updating the posterior distributions with previous computed information being reused in response to small yet frequent changes made to probabilistic knowledge, i.e., prior distributions and observations.

This work has opened up some new research opportunities. We can extend our incremental analysis for probabilistic programs by combining it with traditional incremental analyses for usual programs via demand-driven [31,32] and/or partial program analysis [9,33] in order to also handle the changes made to usual statements. In addition, we can combine our incremental inference with symbolic analysis [10,29] to support incremental symbolic inference with hybrid discrete and continuous distributions being supported.

**Acknowledgments** The authors wish to thank the anonymous reviewers for their valuable comments. The first author would like to thank Aleksandar Chakarov for some helpful email discussions. This work is supported by Australian Research Grants, DP150102109, DP170103956 and DE170101081.

## References

1. S. Arzt and E. Bodden. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, 2014.

2. J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming*, pages 77–96, 2011.
3. A. Caticha, A. Giffin, and A. Mohammad-Djafari. Updating probabilities. In *International Conference on Artificial Intelligence and Pattern Recognition*, pages 31–42, 2006.
4. A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*, pages 153–160, 2013.
5. A. Chakarov and S. Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In *International Static Analysis Symposium*, pages 85–100, 2014.
6. H. Chan and A. Darwiche. On the revision of probabilistic beliefs using uncertain evidence. *Artificial Intelligence*, 163(1):67–90, 2005.
7. S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.
8. G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 92–102, 2013.
9. X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. *26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.
10. T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83, 2016.
11. P. W. Glynn and D. L. Iglehart. Importance sampling for stochastic simulations. *Management Science*, 35(11):1367–1392, 1989.
12. A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181, 2014.
13. W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, 2000.
15. C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 133–144, 2014.
16. D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. 2009.
17. S. Kullback. *Information theory and statistics*. 1997.
18. Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *International Conference on Compiler Construction*, pages 61–81, 2013.
19. D. J. MacKay. Introduction to monte carlo methods. In *Learning in graphical models*, pages 175–204. 1998.
20. P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, 21(4):463–532, 2013.
21. A. C. Miller III and T. R. Rice. Discrete approximations of probability distributions. *Management science*, 29(3):352–362, 1983.

22. T. P. Minka. Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 362–369, 2001.
23. D. Monniaux. Abstract interpretation of probabilistic semantics. In *Proceedings of the 7th International Symposium on Static Analysis*, pages 322–339, 2000.
24. K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.
25. A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *Proceedings of the 29th National Conference on Artificial Intelligence*, pages 2476–2482, 2014.
26. S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.
27. A. Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137:96, 2009.
28. M. Plummer et al. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, page 125, 2003.
29. S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 447–458, 2013.
30. S. Sanner and D. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1384–1390, 2005.
31. L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 270–273, 2012.
32. Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–473, 2016.
33. Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266, 2016.
34. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.
35. M. J. Wainwright, M. I. Jordan, et al. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305, 2008.
36. C. Wanke and D. Greenbaum. Incremental, probabilistic decision making for en route traffic management. *Air Traffic Control Quarterly*, 15(4):299–319, 2007.
37. A. Yue and W. Liu. Revising imprecise probabilistic beliefs in the framework of probabilistic logic programming. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 590–596, 2008.
38. S. Zhan and J. Huang. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 775–786, 2016.