Bayesian Inference Using Data Flow Analysis

Guillaume Claret
INRIA, France
guillaume@claret.me

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Andrew D. Gordon
Microsoft Research
Cambridge
adg@microsoft.com

Johannes Borgström
Uppsala University
johannes.borgstrom@it.uu.se

ABSTRACT

We present a new algorithm for Bayesian inference over probabilistic programs, based on data flow analysis techniques from the program analysis community. Unlike existing techniques for Bayesian inference on probabilistic programs, our data flow analysis algorithm is able to perform inference directly on probabilistic programs with loops. Even for loop-free programs, we show that data flow analysis offers better precision and better performance benefits over existing techniques. We also describe heuristics that are crucial for our inference to scale, and present an empirical evaluation of our algorithm over a range of benchmarks.

1. INTRODUCTION

We present a data flow analysis for probabilistic programs, which can be used to perform Bayesian inference. Before delving into details of the analysis, we first give the reader some background on probabilistic programs and Bayesian inference.

Probabilistic programs are "usual" programs (written in languages like C or Java or LISP or ML) with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program through observations. A variety of probabilistic programming languages and systems have been proposed [11,12,17,19,22,24]. However, unlike "usual" programs which are written for the purpose of being executed, the purpose of a probabilistic program is to implicitly specify a probability distribution. *Bayesian inference* is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program.

Probabilistic programs can be used to represent *probabilistic graphical models* [18], which use graphs to denote conditional dependences between random variables. Probabilistic graphical models are widely used in statistics and machine learning, with diverse application areas including information extraction, speech recognition, computer vision, coding theory, biology and reliability analysis. They allow specification of dependences between random variables via generative models, as well as conditioning of random variables using phenomena or data observed in the real world. A variety of efficient inference algorithms have been developed to analyze and query probabilistic graphical models.

Inference algorithms for probabilistic programs are broadly classified into: (1) dynamic methods such as the Gibbs sampling algorithm, the Metropolis-Hastings (MH) algorithm, which involve executing the program with random draws and computing statistics on the resulting data, and (2) static methods such as message-passing and belief propagation. Current approaches to performing static inference on probabilistic programs involve compiling such programs to graphical models such as Bayesian networks or factor graphs, and using known inference techniques on such models. For instance, in [2], a functional probabilistic program is first translated into a factor graph, and Infer.NET [22] is used to analyze the resulting factor graph and perform inference.

We propose a new direction for efficient static inference of probabilistic programs based on techniques from data flow analysis. Our "data flow facts" are probability distributions, and our analysis merges data flow facts at join points, and computes fixpoints in the presence of loops. However, we show that our data flow analysis does not lose precision, and performs exact Bayesian inference (see Theorem 1).

Our approach is fundamentally different from sampling algorithms, which use multiple concrete executions to represent distributions approximately using a set of samples, and from message-passing algorithms, which maintain representations of approximate distributions. Performing probabilistic inference using data flow analysis offers several advantages. Prior techniques for static inference are restricted to loop-free programs [2, 9]. We are able to statically analyze probabilistic programs with loops using the idea of fixpoints from the program analysis and verification communities. Even for loop-free programs, we show that data flow analysis offers performance benefits over existing techniques. We are able to perform exact inference, and hence compute an answer with better precision than current static techniques which use approximate distributions to scale.

1.1 Probabilistic Programs

We motivate probabilistic programs and inference using a series of examples. Consider Example 1 in Figure 1. Intuitively, this program tosses two fair coins (simulated by drawing from a Bernoulli random variable with mean 0.5),

```
bool c1, c2;
c1 := Bernoulli(0.5);
c2 := Bernoulli(0.5);
```

Example 1.

```
bool c1, c2;
c1 := Bernoulli(0.5);
c2 := Bernoulli(0.5);
observe(c1 || c2);
```

Example 2.

Figure 1: Two probabilistic programs.

```
bool b, c;
b := true;
c := Bernoulli(0.5);
while (c){
  b := !b;
  c := Bernoulli(0.5);
}
```

Example 3.

```
bool c1, c2;
c1 := Bernoulli(0.5);
c2 := Bernoulli(0.5);
while !(c1 || c2) {
   c1 := Bernoulli(0.5);
   c2 := Bernoulli(0.5);
}
```

Example 4.

Figure 2: Probabilistic programs with loops.

```
float skillA, skillB,skillC;
float perfA1,perfB1,perfB2,
      perfC2,perfA3,perfC3;
skillA := Gaussian(100,10);
skillB := Gaussian(100,10);
skillC := Gaussian(100,10);

// first game:A vs B, A won
perfA1 := Gaussian(skillA,15);
perfB1 := Gaussian(skillB,15);
observe(perfA1 > perfB1);
// second game:B vs C, B won
perfB2 := Gaussian(skillB,15);
perfC2 := Gaussian(skillC,15);
observe(perfB2 > perfC2);

// third game:A vs C, A won
perfA3 := Gaussian(skillA,15);
perfC3 := Gaussian(skillC,15);
observe(perfA3> perfC3);
```

Figure 3: (Example 5) TrueSkill skill rating.

assigns the outcomes of these coin tosses to `c1` and `c2` respectively, and returns the values of the two variables `c1` and `c2`. The program represents a probability distribution over Bernoulli variables `c1` and `c2`, where:
Pr(`c1`=`false`,`c2`=`false`) = Pr(`c1`=`false`,`c2`=`true`) = Pr(`c1`=`true`,`c2`=`false`) = Pr(`c1`=`true`,`c2`=`true`) = 1/4.

Next, consider Example 2 in Figure 1. In this program, in addition to tossing the two coins and assigning the outcomes to `c1` and `c2`, we have the statement `observe`(`c1`||`c2`). The semantics of the `observe` statement classifies runs which satisfy the boolean expression `c1`||`c2` as *valid* runs. Runs that do not satisfy `c1`||`c2` are classified as *invalid* runs. The program specifies the generated distribution over the values of the variables (`c1`, `c2`) conditioned over valid runs, which is given by: Pr(`c1`=`false`,`c2`=`false`) = 0, and Pr(`c1`=`false`,`c2`=`true`) = Pr(`c1`=`true`,`c2`=`false`) = Pr(`c1`=`true`,`c2`=`true`) = 1/3.

Next, we consider probabilistic programs with loops. Consider Example 3 in Figure 2. This program initializes `b` to `true` and `c` to the outcome of tossing a coin. Then, it loops until `c` becomes `false`, toggling `b` and assigning to `c` the result from a fresh coin-toss in every iteration of the loop. The while-loop terminates with probability 1, since for the loop to not terminate, `c` should be always assigned `true` from the coin toss, the probability of which decreases exponentially with the number of iterations. The program specifies the generated distribution over the values of the variables (`b`, `c`) given by: Pr(`b`=`true`,`c`=`true`) = 0, and Pr(`b`=`false`,`c`=`true`) = 0, and Pr(`b`=`true`,`c`=`false`) = 2/3, and Pr(`b`=`false`,`c`=`false`) = 1/3.

The probability Pr(`b` = `true`, `c` = `false`) is the probability that the program executes the while loop an even number of times, and is given by the summation (1/2) + (1/8) + (1/32) + ⋯, which equals 2/3. The probability Pr(`b` = `false`, `c` = `false`) is the probability that the program executes the while loop an odd number of times, and is given by the summation (1/4) + (1/16) + (1/64) + ⋯, which equals 1/3.

Consider Example 4 in Figure 2. This program repeatedly assigns to `c1` and `c2` outcomes of fair coin tosses, in a loop, until the condition (`c1`||`c2`) becomes true. Thus, this program specifies the generated distribution over the variables (`c1`, `c2`) given by: Pr(`c1`=`false`,`c2`=`false`) = 0, and Pr(`c1`=`false`,`c2`=`true`) = Pr(`c1`=`true`,`c2`=`false`) = Pr(`c1`=`true`,`c2`=`true`) = 1/3. The alert reader would notice that this distribution is identical to the distribution specified by Example 2. Though `observe` statements can be equivalently represented using `while` loops using a simple program transformation illustrated in this example, our inference algorithm handles `observe` statements more efficiently, when compared to loops. Also, there is no simple transformation that converts any `while` loop to an `observe` statement. Consequently, we have both `observe` statements and `while` loops in our language.

The final example we use in this introduction is a simplified version of the TrueSkill [14] skill rating system used by Microsoft's Xbox Live to rate the relative skills of players playing online games. In Example 5 (Figure 3), we have 3 players, A, B and C, whose skills are given by variables `skillA`, `skillB` and `skillC` respectively, which are initialized by drawing from a Gaussian distribution with mean 100, and variance 10. Based on the outcomes of some number of played games (which is 3 games in this example), we condition the skills thus generated. The first game was played between A and B, and A won the game. This is modeled by assigning to the two random variables `perfA1` and `perfB1` denoting the performance of the two players in the first game, and constraining that `perfA1` is greater than `perfB1` using an `observe` statement. Note that the performance of a player (such as player A) is a function of her skill, but with additional Gaussian noise introduced in order to model the variation in performance we may see because of incompleteness in our model (such as the amount of sleep the player got the previous night). The second game was played between B and C, and B won the game. The third game was played between A and C, and A won the game. Using this model, we want to calculate the joint probability distribution of these random variables, and use this to estimate the relative skills of the players. Note that each `observe` statement constrains performances in a game, and implicitly the skills of the players, since performances depend on skills. Such a rating can be used to give points, or match players having comparable skill for improved gaming experience. In this example, the skills `skillA`, `skillB`, and `skillC` inferred by our tool are: `skillA` = `Gaussian`(102.1, 7.8), `skillB` = `Gaussian`(100.0, 7.6), `skillC` = `Gaussian`(97.9, 7.8). Note that since A won against both B and C, and B won against C, the resulting distribution agrees with our intuitive assess-

ment of their relative skills.

1.2 Inference

Calculating the distribution specified by a probabilistic program is called *inference*. The inferred probability distribution is called *posterior probability* distribution, and the initial guess made by the program is called the *prior probability* distribution. For instance, in Example 5, the prior distribution for `skillA` is `Gaussian`(100,10), whereas the posterior distribution is `Gaussian`(102.1, 7.8). One way to perform inference is runtime execution. We can execute the program several times using sampling to execute probabilistic statements, and observe the values of the desired variables in valid runs [12], and compute statistics on the data to infer an approximation to the desired distribution. Alternatively, a probabilistic program can be compiled to a graphical model [2,19] over which inference is performed using message passing algorithms such as belief propagation and its variants [23].

Data flow analysis, invented by Kildall [16], uses a lattice of data flow facts, merging at join points, and fixpoints for loops to compute solutions to several "meet-over-all-paths" (MOP) analysis problems. The main contribution of this paper is a new technique to perform inference on probabilistic programs using data flow analysis.

We consider Boolean Probabilistic Programs, and in particular the programming language BernoulliProb (see Section 3), where all variables are boolean and the only distribution allowed is the Bernoulli distribution. Our main formal result, Theorem 1, is the correctness of our inference algorithm with respect to the formal semantics of our probabilistic language.

Probabilistic programs with discrete variables over a finite domain can be directly encoded with BernoulliProb, without any approximation. Further, probabilistic programs with continuous variables can be approximated to boolean programs by approximating continuous distributions with discrete distributions (see Section 4). Using such transformations, any probabilistic program can be approximately represented in BernoulliProb and analyzed using our technique.

We have implemented our approach and we find that in several examples, we are able to perform exact inference. If the probabilistic program has a large number of variables, explicit representation of joint probability distributions is expensive. Our implementation uses Algebraic Decision Diagram (ADD) [1], a graphical data structure for compactly representing finite functions, to represent probability distributions, and perform data flow analysis symbolically. For large examples, where exact inference is infeasible, we propose a *batching technique* where we periodically project the joint distribution to marginal distributions over individual variables, to save space at the cost of some approximation. In our experiments, our approximate inference produces results with better precision than other state-of-the-art inference approaches.

To summarize, our main contributions are as follows:

- We present a new Bayesian inference algorithm for probabilistic programs that is based on data flow analysis. Our algorithm merges data flow facts (which are probability distributions in our case) at join points, avoiding exponential explosion in exploring all paths. Using the notion of fixpoints, we are able to handle programs with loops, which current techniques such as message passing on Bayesian networks are not able to handle.

- We present techniques to make our algorithm work on continuous distributions (via discretization), and large programs (via batching and other heuristics).

- We present empirical results from running the exact and approximate versions of our algorithm on various benchmarks.

This work is related thematically to our other recent work in the intersection of program analysis and Bayesian inference. In particular, related recent efforts include using weakest preconditions to perform efficient sampling [5] and using a framework of model-learners to do Bayesian reasoning [13].

r	$\in$	$\mathbb{R}$	
x	$\in$	Vars	
$\mathcal{T}$	::=	`bool`	types
uop	::=	`not`	unary operators
bop	::=	`and` | `or`	binary operators
$\mathcal{D}$	::=	$\mid \mathcal{T}\, x_1, x_2, \ldots, x_n$	declaration
$\mathcal{E}$	::=		expressions
		x	variable
		$\mid c$	constant
		$\mid \mathcal{E}_1$ **bop** $\mathcal{E}_2$	binary operation
		$\mid$ **uop** $\mathcal{E}_1$	unary operation
$\mathcal{S}$	::=		statements
		$x := \mathcal{E}$	deterministic assignment
		$\mid x :=$ `Bernoulli`(r)	Bernoulli assignment
		$\mid$ `observe` $(\mathcal{E})$	observe
		$\mid$ `skip`	skip
		$\mid \mathcal{S}_1; \mathcal{S}_2$	sequential composition
		$\mid$ `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$	conditional composition
		$\mid$ `while` $\mathcal{E}$ `do` $\mathcal{S}_1$	loop
$\mathcal{P}$	::=	$\mathcal{D}\ \mathcal{S}$	programs

Figure 4: Syntax of BernoulliProb.

2. PROBABILISTIC PROGRAMS

We start by formally defining boolean probabilistic programs. Figure 4 shows the syntax of BernoulliProb. The only type T allowed in the language is the boolean type, with values `true` and `false`. A program has a declaration of variables $x_1, x_2, \ldots, x_n$ followed by statements. We use $\mathcal{V}(P)$ to denote the variables of program P, and $\mathcal{S}(P)$ to denote the statement body of program P. Primitive statements include deterministic assignments, Bernoulli assignment, observe and skip statements. A deterministic assignment is of the form $x := \mathcal{E}$, where $\mathcal{E}$ is an expression. Expressions are formed from variables and constants using binary and unary operators. A Bernoulli assignment is of the form $x :=$ `Bernoulli`(r), where r is a real number. An observe statement is of the form `observe`$(\mathcal{E})$, where $\mathcal{E}$ is an expression. A skip statement is of the form `skip`. Compound statements are formed from primitive statements using sequential composition, conditional composition and looping.

$\langle \sigma, x := \mathcal{E} \rangle \rightarrow^1 \langle \sigma[x \leftarrow \sigma(\mathcal{E})], \texttt{skip} \rangle$
$\langle \sigma, x := \texttt{Bernoulli}(r) \rangle \rightarrow^r \langle \sigma[x \leftarrow \texttt{true}], \texttt{skip} \rangle$
$\langle \sigma, x := \texttt{Bernoulli}(r) \rangle \rightarrow^{1-r} \langle \sigma[x \leftarrow \texttt{false}], \texttt{skip} \rangle$
$\langle \sigma, \texttt{observe}(\mathcal{E}) \rangle \rightarrow^1 \langle \sigma, \texttt{skip} \rangle$, if $\sigma(\mathcal{E}) = \texttt{true}$
$\langle \sigma, \texttt{skip}; \mathcal{S} \rangle \rightarrow^1 \langle \sigma, \mathcal{S} \rangle$
$\langle \sigma, \mathcal{S}_1; \mathcal{S}_2 \rangle \rightarrow^p \langle \sigma', \mathcal{S}'; \mathcal{S}_2 \rangle$, if $\langle \sigma, \mathcal{S}_1 \rangle \rightarrow^p \langle \sigma', \mathcal{S}' \rangle$
$\langle \sigma, \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \rangle \rightarrow^1 \langle \sigma, \mathcal{S}_1 \rangle$, if $\sigma(\mathcal{E}) = \texttt{true}$
$\langle \sigma, \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \rangle \rightarrow^1 \langle \sigma, \mathcal{S}_2 \rangle$, if $\sigma(\mathcal{E}) = \texttt{false}$
$\langle \sigma, \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S} \rangle \rightarrow^1 \langle \sigma, \texttt{skip} \rangle$, if $\sigma(\mathcal{E}) = \texttt{false}$
$\langle \sigma, \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S} \rangle \rightarrow^1 \langle \sigma, \mathcal{S}; \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S} \rangle$, if $\sigma(\mathcal{E}) = \texttt{true}$

Figure 5: Semantics of BernoulliProb.

The operational semantics of BernoulliProb is given in Figure 5 as a probabilistic transition system. A *state* σ of the program with variables $x_1, x_2, \ldots, x_n$ is a valuation to all the variables. The domain of all possible states is Γ. A configuration ω is a pair $\langle \sigma, \mathcal{S} \rangle$, where σ is a state, and $\mathcal{S}$ is a statement. Intuitively, a run of a program returns the final state σ', which is the first component of the configuration on termination. Since programs are probabilistic, each time we run a program P we might get a different final state. The semantics of the program is the distribution over final states returned by the program. We formalize this below.

Given a state σ, we use the notation $\sigma(x_i)$ to denote the value of variable x_i in σ, and the notation $\sigma(\mathcal{E})$ to denote the value of the expression $\mathcal{E}$ in σ.

The probabilistic transition system shown in Figure 5 has transition rules of the form form $\omega_1 \rightarrow^p \omega'$, meaning configuration ω takes a step to configuration ω' with probability p, inspired by the transition system for the functional language Fun [2]. We use the notation $\sigma[x \leftarrow v]$ for the state obtained by updating the value of x in σ with v and leaving the values of all other variables in σ unchanged. The only transition in Figure 5 whose probability is not 1, is the one for Bernoulli assignment $x := \texttt{Bernoulli}(r)$. This statement assigns `true` to x with probability r and, `false` to x with probability $1 - r$. The configuration $\langle \sigma, \texttt{observe}(\mathcal{E}) \rangle$ transitions to the configuration $\langle \sigma, \texttt{skip} \rangle$ with probability 1 if $\sigma(\mathcal{E})$ is equal to `true`. In this case, we say that the observation *succeeds*. Otherwise, if $\sigma(\mathcal{E})$ is equal to `false`, then the configuration $\langle \sigma, \texttt{observe}(\mathcal{E}) \rangle$ gets stuck with no outgoing transitions. Thus, implicitly, the resulting distribution of a program is conditioned on all observations succeeding.

The sequential composition $\mathcal{S}_1; \mathcal{S}_2$ transitions depending on the transition of the first statement $\mathcal{S}_1$. The conditional composition `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$ transitions according to how the current state σ evaluates the expression $\mathcal{E}$. The while loop `while` $\mathcal{E}$ `do` $\mathcal{S}$ also transitions according to how the current state σ evaluates the expression $\mathcal{E}$. If the expression evaluates to `false` the loop exits, otherwise it executes the body $\mathcal{S}$ and loops.

A *run* of a statement $\mathcal{S}$ starting from state σ and ending in state σ' is a sequence $\omega = (\omega_1, \omega_2, \ldots, \omega_{n+1})$ for $n \geq 0$, where the following conditions are satisfied: (1) $\omega_1 \rightarrow^{p_1} \omega_2 \cdots \omega_n \rightarrow^{p_n} \omega_{n+1}$, (2) the initial configuration $\omega_1 = \langle \sigma, \mathcal{S} \rangle$, and (3) the final configuration $\omega_{n+1} = \langle \sigma', \texttt{skip} \rangle$, where the statement part is equal to `skip`, signifying termination of execution. Given such a run ω, we say that the statement S evaluates to state σ' starting at state σ with probability $\Pr(\omega) = p_1 \cdots p_n$. The set of all runs of the statement $\mathcal{S}$ starting from state σ and ending with state σ' is denoted by $\Omega(\sigma, \mathcal{S}, \sigma')$. Note that we are only concerned with runs of finite length in $\Omega(\sigma, \mathcal{S}, \sigma')$ that end in a configuration with a `skip` statement. Since the language BernoulliProb has loops, the number of such runs is potentially infinite.

Given a statement $\mathcal{S}$, the probability $\Pr(\sigma, \mathcal{S}, \sigma')$ of executing statement $\mathcal{S}$ starting at state σ and ending at state σ' is given by

$$\Pr(\sigma, \mathcal{S}, \sigma') = \Sigma_{\omega \in \Omega(\sigma, \mathcal{S}, \sigma')} \Pr(\omega)$$

Note that even though the summation is potentially over an infinite number of runs, we are interested in its limit which is always lesser than one. Given a distribution ρ over program states Γ, we define

$$\Pr(\rho, \mathcal{S}, \sigma') = \Sigma_{\sigma \in \Gamma} \rho(\sigma) \cdot \Pr(\sigma, \mathcal{S}, \sigma')$$

Consider a program P with variables $x_1, x_2, \ldots, x_n$ and statement $\mathcal{S}$. Let σ_0 denote the state where all variables $x_1, x_2, \ldots, x_n$ are assigned `false`, and ρ_0 denote the Dirichlet distribution over states $\lambda\sigma.\texttt{ite}(\sigma = \sigma_0, 1, 0)$, where the expression $\texttt{ite}(e, x, y)$ evaluates to x if e is `true`, and y if e is `false`. Intuitively, the semantics of the program P is the probability distribution ρ obtained by starting the execution of $\mathcal{S}$ from initial state σ_o. Formally, the semantics of P is the distribution $\lambda\sigma.\Pr(\rho_0, \mathcal{S}, \sigma)$.

We note that the semantics given here sums over all the runs of the program. In Section 3, we present our main result of the paper, which shows that this summation can be computed by a data flow analysis which merges data flow facts at join points.

We end this section by noting two facts about the semantics of BernoulliProb programs. First, if a program P has non-trivial `observe` statements, the resulting distribution $\lambda\sigma.\Pr(\rho_0, \mathcal{S}, \sigma)$ is not necessarily normalized (i.e., the sum of the probabilities over all states can be strictly less than 1). If we wanted a distribution, we can calculate the sum of probabilities S of all the states and appropriately normalize each of the probabilities by S. Second, we can write pathological programs containing statements such as `observe(false)` or non-terminating while loops with no terminating executions. The output distribution of such a program maps all states to 0 probability, which is equivalent to saying that the semantics of such a program is undefined.

Algorithm 1 The Infer algorithm.

Algorithm Infer
Input: A BernoulliProb program P.
Output: A posterior distribution ρ over P's output.
1: $\rho_0 := \lambda\sigma.\texttt{ite}(\forall x_i \in \mathcal{V}(P).\sigma(x_i) = \texttt{false}, 1, 0)$;
2: $\rho := \textsc{Post}(\rho_0, \mathcal{S}(P))$
3: $\rho_N := \textsc{Normalize}(\rho)$
4: **return** ρ_N

3. ALGORITHM

Algorithm 1 describes the inference algorithm Infer for BernoulliProb programs, which is based on data flow analysis. Infer takes a BernoulliProb program P as input and returns the joint distribution over the output states of P (see Section 2). Line 1 constructs an initial distribution ρ_0, which is a Dirichlet distribution mapping the state with every variable set to `false` as having probability 1,

Algorithm 2 The POST computation.

Algorithm $\textsc{Post}(\rho, \mathcal{S})$

Input: An input distribution ρ over the states of the program P, and a statement $\mathcal{S}$

Output: Output distribution over the states of the program P

1: **switch** $(\mathcal{S})$
2: **case** $x := \mathcal{E}$**:**
3: **return** $\lambda\sigma.\Sigma_{\{\sigma' | \sigma'[x \leftarrow \sigma'(\mathcal{E})] = \sigma\}} \rho(\sigma')$
4: **case** $x :=$ `Bernoulli`(r)**:**
5: **return** $\lambda\sigma.(r \times \Sigma_{\{\sigma' | \sigma'[x \leftarrow \texttt{true}] = \sigma\}} \rho(\sigma') +$
6: $(1 - r) \times \Sigma_{\{\sigma' | \sigma'[x \leftarrow \texttt{false}] = \sigma\}} \rho(\sigma'))$
7: **case** `observe` $(\mathcal{E})$**:**
8: **return** $\lambda\sigma.\texttt{ite}(\sigma(\mathcal{E}), \rho(\sigma), 0)$
9: **case** `skip`**:**
10: **return** ρ
11: **case** $\mathcal{S}_1; \mathcal{S}_2$**:**
12: $\rho' = \textsc{Post}(\rho, \mathcal{S}_1)$;
13: **return** $\textsc{Post}(\rho', \mathcal{S}_2)$
14: **case** `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$**:**
15: $\rho_t = \lambda\sigma.\texttt{ite}(\sigma(\mathcal{E}), \rho(\sigma), 0)$;
16: $\rho_f = \lambda\sigma.\texttt{ite}(\sigma(\mathcal{E}), 0, \rho(\sigma))$;
17: **return** $\lambda\sigma.(\textsc{Post}(\rho_t, \mathcal{S}_1)(\sigma) + \textsc{Post}(\rho_f, \mathcal{S}_2)(\sigma))$
18: **case** `while` $\mathcal{E}$ `do` $\mathcal{S}_1$ **:**
19: $\rho_p := \bot; \rho_c := \rho$
20: **while** $\rho_p \neq \rho_c$ **do**
21: $\rho_p := \rho_c$
22: $\rho_c := \textsc{Post}(\rho_p,$ `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else skip`$)$
23: **end while**
24: **return** ρ_c
25: **end switch**

and all other states as having probability 0. Recall that the function $\texttt{ite}(e, x, y)$ evaluates to x if $e =$ `true` and y if $e =$ `false`. The procedure POST (line 2) returns the posterior sub-distribution over the output of statements $\mathcal{S}(P)$ of program P, starting with ρ_0 as the input distribution. POST is a recursive procedure shown in Algorithm 2.

The function NORMALIZE takes the output of POST, which is a function ρ from output values of P to $[0, 1]$, and if $\text{range}(\rho) \neq \{0\}$ it returns a probability distribution ρ_N over output values, given by $(1/\Sigma_{v \in \Gamma} \rho(v)) \cdot \rho$ (recall that Γ is the domain of all possible states).

Algorithm 2 relies on notations introduced in Section 2 for valuations σ, $\sigma[x \leftarrow v]$, and values $\sigma(x)$, $\sigma(\mathcal{E})$. Distributions ρ, ρ_c, etc (normalized or un-normalized) map valuations to $[0, 1]$. Let $\bot$ denote the null map which maps every valuation to 0.

POST operates recursively over the syntax of an input BERNOULLIPROB statement $\mathcal{S}$ (as defined in Figure 5). In lines 2–3, POST handles the case when the statement is a deterministic assignment. In this case, the output distribution maps a state σ to the sum over all the input densities of states σ' that equal σ on executing the deterministic assignment. Lines 4–6 handle Bernoulli assignment. The output distribution for this statement is a convex combination of the result of the deterministic assignment $x :=$ `true` scaled by r, and the deterministic assignment $x :=$ `false` scaled by $1 - r$. Lines 7–8 handle the observe statement. The output distribution (which is unnormalized) maps a state σ to the density over the input distribution if the expression $\mathcal{E}$ evaluates to `true` and 0 otherwise. Note that the output distribution here is unnormalized, unless $\mathcal{E}$ is `true` for all states. Lines 9–10 handle the skip statement, which is an identity for POST.

Lines 11–12 handle sequential composition by first computing POST over the first statement, and using the resulting distribution to compute POST over the second statement. Lines 14–17 handle conditional statements. The output distribution is the pointwise sum of the distribution obtained from the "if-part" and the "else-part". The "if-part" and "else-part" are recursively computed by applying POST on their bodies, after splitting the input distribution ρ depending on the condition predicate $\mathcal{E}$ to ρ_t and ρ_f.

The final case (lines 18–24) handles a while loop by computing a fixpoint. It uses two scratch variables ρ_p and ρ_c to represent the "previous" distribution and "current" distribution respectively. Iteratively, POST is repeatedly applied on the input distribution ρ_p with the statement `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else skip` until the output distribution ρ_c is the same as the input ρ_p. We note that this fixpoint is potentially nonterminating, even though ρ_p and ρ_c may converge in the limit. In our implementation we terminate the fixpoint when the KL-divergence between ρ_c and ρ_p goes below a certain threshold.

Readers familiar with data flow analysis will note that our data flow facts are probability distributions, and our algorithm merges these distributions at join points (lines 14–17), and computes fixpoints for loops (lines 18–24). Next, we prove that even with such merging of data flow facts at join points, this algorithm computes the exact posterior distribution given by the "sum over all paths" definition in Section 2.

LEMMA 1. *For any statement $\mathcal{S}$ and any distribution ρ, if the* POST *algorithm terminates, then:*

$$\lambda\sigma.\Pr(\rho, \mathcal{S}, \sigma) = \textsc{Post}(\rho, \mathcal{S})$$

Proof: We show that for any statement $\mathcal{S}$ over variables $x_1, x_2, \ldots, x_n$, any input distribution ρ over the program states, and any output state σ, we have that $\Pr(\rho, \mathcal{S}, \sigma) = \textsc{Post}(\rho, \mathcal{S})(\sigma)$ when the POST algorithm terminates. The proof is by induction over the structure of $\mathcal{S}$, and we carry it out by performing a case analysis of all types of statements supported in BERNOULLIPROB.

Case 1(deterministic assignment):
If $\mathcal{S} = x := \mathcal{E}$, we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \sigma = \sigma_1[x \leftarrow \sigma_1(\mathcal{E})]\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) + \\
&\quad \Sigma_{\{\sigma_1 \in \Gamma | \sigma \neq \sigma_1[x \leftarrow \sigma_1(\mathcal{E})]\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \sigma = \sigma_1[x \leftarrow \sigma_1(\mathcal{E})]\}} \rho(\sigma_1) \times 1 + \\
&\quad \Sigma_{\{\sigma_1 \in \Gamma | \sigma \neq \sigma_1[x \leftarrow \sigma_1(\mathcal{E})]\}} \rho(\sigma_1) \times 0 \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \sigma = \sigma_1[x \leftarrow \sigma_1(\mathcal{E})]\}} \rho(\sigma_1) \\
&= \textsc{Post}(\rho, x := \mathcal{E})(\sigma) \\
&\qquad\qquad \text{(from line 3 of Algorithm 2)}
\end{aligned}$$

Case 2(Bernoulli assignment):
If $\mathcal{S} = x :=$ `Bernoulli`(r), we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \sigma = \sigma_1[x \leftarrow \texttt{true}]\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) + \\
&\quad \Sigma_{\{\sigma_1 \in \Gamma | \sigma \neq \sigma_1[x \leftarrow \texttt{false}]\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \sigma = \sigma_1[x \leftarrow \texttt{true}]\}} \rho(\sigma_1) \times r + \\
&\quad \Sigma_{\{\sigma_1 \in \Gamma | \sigma \neq \sigma_1[x \leftarrow \texttt{false}]\}} \rho(\sigma_1) \times (1 - r) \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \textsc{Post}(\rho, x := \texttt{Bernoulli}(r))(\sigma) \\
&\qquad\qquad \text{(from lines 5–6 of Algorithm 2)}
\end{aligned}$$

Case 3(Observe statement):
If $\mathcal{S} = \texttt{observe}(\mathcal{E})$, we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{true}\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) + \\
&\qquad \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{false}\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{true}\}} \rho(\sigma_1) \times \texttt{ite}(\sigma = \sigma_1, 1, 0) + \\
&\qquad \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{false}\}} \rho(\sigma_1) \times 0 \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \texttt{ite}(\mathcal{E}(\sigma), \rho(\sigma), 0) \\
&= \textsc{Post}(\rho, \texttt{observe}(\mathcal{E}))(\sigma) \\
&\qquad\qquad \text{(from line 8 of Algorithm 2)}
\end{aligned}$$

Case 4(Skip statement):
if $\mathcal{S} = \texttt{skip}$, we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \rho(\sigma) \times \Pr(\sigma, \mathcal{S}, \sigma) + \\
&\qquad \Sigma_{\{\sigma_1 \in \Gamma | \sigma_1 \neq \sigma\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \rho(\sigma) \times 1 + \Sigma_{\{\sigma_1 \in \Gamma | \sigma_1 \neq \sigma\}} \rho(\sigma_1) \times 0 \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \rho(\sigma) \\
&= \textsc{Post}(\rho, \texttt{skip})(\sigma) \\
&\qquad\qquad \text{(from line 10 of Algorithm 2)}
\end{aligned}$$

Case 5(Sequential composition):
If $\mathcal{S} = \mathcal{S}_1; \mathcal{S}_2$, we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Sigma_{\sigma_2 \in \Gamma} \Pr(\sigma_1, \mathcal{S}_1, \sigma_2) \times \Pr(\sigma_2, \mathcal{S}_2, \sigma) \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \Sigma_{\sigma_1 \in \Gamma} \Sigma_{\sigma_2 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}_1, \sigma_2) \times \Pr(\sigma_2, \mathcal{S}_2, \sigma) \\
&= \Sigma_{\sigma_2 \in \Gamma} \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}_1, \sigma_2)) \times \Pr(\sigma_2, \mathcal{S}_1, \sigma) \\
&= \Sigma_{\sigma_2 \in \Gamma} \Pr(\rho, \mathcal{S}_1, \sigma_2) \times \Pr(\sigma_2, \mathcal{S}_1, \sigma) \\
&= \Sigma_{\sigma_2 \in \Gamma} \textsc{Post}(\rho, \mathcal{S}_1)(\sigma_2) \times \Pr(\sigma_2, \mathcal{S}_1, \sigma) \\
&\qquad\qquad \text{(by induction)} \\
&= \Pr(\textsc{Post}(\rho, \mathcal{S}_1), \mathcal{S}_2, \sigma) \\
&= \textsc{Post}(\rho, \mathcal{S}_1; \mathcal{S}_2)(\sigma) \\
&\qquad\qquad \text{(from lines 12–13 of Algorithm 2)}
\end{aligned}$$

Case 6(Conditional composition):
If $\mathcal{S} = \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2$ we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{true}\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}_1, \sigma) + \\
&\qquad \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{false}\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}_2, \sigma) \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \Sigma_{\sigma_1 \in \Gamma} \texttt{ite}(\mathcal{E}(\sigma_1), \rho(\sigma_1), 0) \times \Pr(\sigma_1, \mathcal{S}_1, \sigma) + \\
&\qquad \Sigma_{\sigma_1 \in \Gamma} \texttt{ite}(\mathcal{E}(\sigma_1), 0, \rho(\sigma_1)) \times \Pr(\sigma_1, \mathcal{S}_2, \sigma) \\
&= \textsc{Post}(\lambda\sigma_1.\texttt{ite}(\mathcal{E}(\sigma_1), \rho(\sigma_1), 0), \mathcal{S}_1)(\sigma) + \\
&\qquad \textsc{Post}(\lambda\sigma_1.\texttt{ite}(\mathcal{E}(\sigma_1), 0, \rho(\sigma_1)), \mathcal{S}_2)(\sigma) \\
&\qquad\qquad \text{(by induction)} \\
&= \textsc{Post}(\rho, \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2)(\sigma) \\
&\qquad\qquad \text{(from lines 15–17 of Algorithm 2)}
\end{aligned}$$

Case 7(While loop):
If $\mathcal{S} = \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}_1$ we have that

$$\begin{aligned}
&\Pr(\rho, \mathcal{S}, \sigma) = \Sigma_{\sigma_1 \in \Gamma} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}, \sigma) \\
&= \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{false}\}} \rho(\sigma_1) + \\
&\qquad \Sigma_{\{\sigma_1 \in \Gamma | \mathcal{E}(\sigma_1) = \texttt{true}\}} \rho(\sigma_1) \times \Pr(\sigma_1, \mathcal{S}_1; \mathcal{S}, \sigma) \\
&\qquad\qquad \text{(from Figure 5)} \\
&= \Pr(\rho, (\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else skip}); \mathcal{S}, \sigma) \\
&= \textsc{Post}(\rho, \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}_1)(\sigma) \\
&\qquad\qquad \text{(from lines 19–24 of Algorithm 2)}
\end{aligned}$$

■

Our main theorem, which states that the data flow analysis algorithm computes the exact posterior distribution as specified by the "sum over all paths" semantics is stated below, and follows directly from the above lemma.

THEOREM 1. *Let P be a* BERNOULLIPROB *program with n variables $x_1, x_2, \ldots, x_n$ and a statement S. Let σ_o be a state which assigns all n variables the value* `false`*, and let ρ_o be the Dirichlet distribution which maps such a state σ_o to probability 1 and all other states to the probability 0. Then, if the* POST *algorithm terminates, then the output distribution of P is given by the* INFER *algorithm.*

4. DISCRETIZATION

We extend BERNOULLIPROB to handle continuous distributions. There are two phases. First, we extend BERNOULLIPROB to support discrete distributions over finite sets. Next, we show how we can approximate a continuous distribution as a discrete distribution over a finite set, which effectively shows that a probabilistic program defined over continuous distributions can be approximated by a BERNOULLIPROB program.

Let D be a distribution over elements of a finite set S of cardinality $|S| > 0$. Then, we can encode elements of S using tuples of $\log |S|$ boolean variables in the standard way (that is, using boolean tuples in $\{0,1\}^{\log |S|}$. Thus, using this encoding, we are able to model D as a distribution over tuples of boolean variables, thus reducing a program defined over arbitrary finite distributions to a BERNOULLIPROB program.

Now consider a continuous distribution $\mathcal{N}$. For ease of exposition, we assume that $\mathcal{N}$ is a Gaussian distribution whose probability density function is defined as follows

$$f(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ and σ^2 are the mean and variance parameters respectively. For a suitable choice $i \in \mathbb{N}$, define an interval $[a, b]$ such that $a = \mu - i\sigma$ and $b = \mu + i\sigma$. For some $w > 0$, define the following set.

$$S = \left\{ a + k\frac{b-a}{w} \mid 0 \leq k < w \right\}$$

We define a discrete distribution D over the elements of S which is a discrete probability mass function that approximates the Gaussian probability density function $f(x; \mu, \sigma^2)$ as follows.

$$D(x \in S; i, w) = \frac{1}{N} \int_x^{x + \frac{b-a}{w}} f(x; \mu, \sigma^2) dx$$

where i and w are parameters that control the degree of approximation, and

$$N = \int_a^b f(x; \mu, \sigma^2) dx$$

a normalization constant ensuring that D is a probability distribution.

It is easy to see that the degree of approximation is controlled by the parameters i and w. In particular, the approximation improves with $i \to \infty$ and $w \to 0$.

Therefore, with the two reductions described above, we are able to reduce probabilistic programs defined over continuous and finite distributions to BernoulliProb programs.

5. IMPLEMENTATION

In this section we describe data structures that we use to represent probability distributions, which are our data flow facts, and details on how we implement our inference algorithm using these data structures. We also present heuristics we use to scale our implementation to large probabilistic programs.

Algebraic Decision Diagrams. Recall that our algorithm (in Section 3) maintains joint probability distributions at every program point. A probability distribution is a real valued function over variable values. For example, after executing the fragment $x = \texttt{Bernoulli}(0.5)$, we obtain the function $\lambda x.0.5$ representing the distribution which maps both values of x (`true` and `false`) to 0.5. As a second example, after symbolically executing Example 2 (Figure 1), we obtain the function $\lambda(c1, c2).\texttt{ite}((c1||c2), 1/3, 0)$.

These are functions from tuples of boolean values to real numbers. One way to represent such functions is using tables (similar to truth tables, but having probabilities as the range). However, with n boolean variables a table representation has 2^n rows, and this is infeasible for large n. Algebraic Decision Diagrams (ADDs) can compactly represent such functions as directed acyclic graphs. ADDs [1] are generalizations of Binary Decision Diagrams (BDDs), invented by Bryant [4]. An ADD is a directed acyclic graph. Each internal node of the graph is a decision node, labeled with a variable name. Each leaf node is labeled with a real value. Each internal node n has two outgoing edges labeled with 0 and 1 respectively, and the target of these edges are called 0-successor and 1-successor of n respectively. Each ADD fixes a total order among its variables, and the ordering of variables in every path respects this total order. Furthermore, each variable occurs at most once on a path from the root to the leaf. ADDs can be compactly constructed from decision trees by performing the following two reductions until saturation: (1) merging isomorphic nodes, and (2) eliminating nodes whose 0-successor and 1-successor are identical. Once we fix a total ordering of variables, and apply the above 2 reductions until they can no longer be applied, ADD is canonical (regardless of the order in which the reductions were applied).

Functions can be manipulated using graph algorithms on their ADD representations. For example, if f_1 and f_2 are two functions represented as ADDs with sizes $|f_1|$ and $|f_2|$ respectively, the ADD for operations such as $f_1 + f_2$ or $f_1 \times f_2$ can be obtained using graph algorithms on the ADDs of f_1 and f_2 with a worst-case complexity of $O(|f_1| \cdot |f_2|)$. Given a function f with a free variable x represented as an ADD, we can also obtain the ADD for $\exists x.f$ by eliminating x from the ADD using graph operations, and potentially doubling the size of the ADD in the worst case. We refer the reader to [1] for details of these algorithms.

For instance, Figure 6 shows the ADD for the distribution of Example 2 from Figure 1: part (a) shows the distribution represented as a decision tree, and part (b) shows the ADD obtained by applying the two reductions above until none applies.

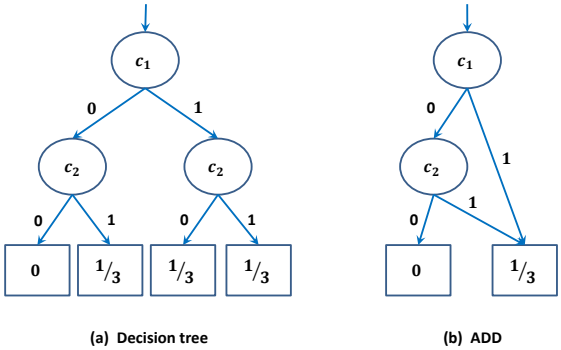

Figure 6: ADD representation of Example 2.

Our implementation of the inference algorithm uses ADDs for a compact representation over distributions. Each of the operations in the Algorithm 2 can be implemented using ADD operations as described below. Consider the operation for processing the deterministic assignment statement in line 3. Let f_ρ be the ADD representation of ρ and let $f_{x'=\mathcal{E}}$ be the ADD representing the relation $x' = \mathcal{E}$, where x' is a fresh variable. We implement the summation in line 3 as follows: (1) First we compute $g = f_\rho \wedge f_{x'=\mathcal{E}}$. (2) Next we existentially quantify x from g to get $h = \exists x.g$. (3) Finally, we rename x' to x in h to get the result of Post as $h[x'/x]$ and return the resulting ADD (we note that $\wedge$, existential quantification and renaming are implemented by ADD packages using graph operations). The implementation for Bernoulli assignment can be done by separately processing the two deterministic assignments $x :=$ `true` and $x :=$ `false` as above, and scaling the two resulting ADDs by r and $(1 - r)$ respectively, and adding them (we note that scaling and adding operations provided by ADD packages). The operations in lines 8, 15 and 16 can be directly implemented on ADDs since `ite` operation is supported by ADD packages. The fixpoint computation in lines 20–23 can be implemented using the techniques described above, and in addition we terminate the fixpoint when the KL-divergence between ρ_c and ρ_p goes below a certain threshold.

Each of these operations $\wedge$, `ite`, scaling, summation, equality check are directly supported by ADD packages such as CUDD [27] and takes time proportional to the product of the sizes of the arguments in the worst case. The normalization operation in Algorithm 1 is not directly supported by ADD packages. We implement this operation using a bottom-up scan of the ADD in time proportional to the size of the ADD.

Figure 7 illustrates how the Infer algorithm proceeds on Example 2 from Figure 1. In particular, the Post computation on the first two statements $\texttt{c1} := \texttt{Bernoulli}(0.5)$ and $\texttt{c2} := \texttt{Bernoulli}(0.5)$ results in uniform distributions over `c1` and `(c1, c2)` respectively. Both these distributions are compactly represented by ADDs with a single leaf node. Next, Post processes the statement $\texttt{observe}(\texttt{c1}||\texttt{c2})$ which results in a subdistribution represented by the ADD shown

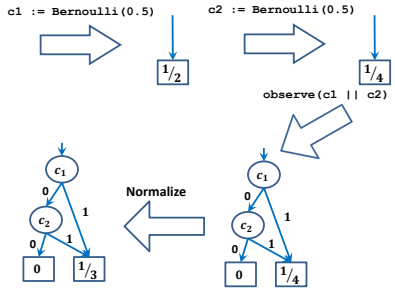

Figure 7: Sequence of ADDs obtained by applying Post and NORMALIZE to Example 2.

in the figure. Finally, INFER normalizes this subdistribution (via the call to NORMALIZE) in order to obtain the final ADD representing the posterior distribution of $(\texttt{c1}, \texttt{c2})$.

Our implementation uses the ADD library from the CUDD package [27]. Our implementation supports a much richer language than BERNOULLIPROB, including continuous distributions, floating point variables, multidimensional array with statically determined sizes, for-loops, etc. Continuous distributions are automatically discretized using the technique in Section 5, and Algorithm 2 is easily extended to for-loops and static sized arrays.

Heuristics. We have also implemented heuristics for scaling and optimizing our implementation. These are as follows:

- *Fast exponentiation*: To speed up the POST computation over while loops we employ exponentiation. We describe exponentiation by way of an example. Consider the program $\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}$ and an input distribution ρ_0. Define $f(\rho) = \text{POST}(\rho, (\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else skip}))$. Then the POST algorithm computes the posterior distribution by applying f to ρ_0 until a fixpoint is reached (in practice, a fixed constant number of times). If we can symbolically represent f, then we can efficiently compute f^{2^n} by computing $f^2 = f \circ f$, $f^4 = f^2 \circ f^2, \ldots$, and finally evaluate $f^{2^n}(\rho_0)$. This can be done as follows:

$$f(\rho) = \lambda\sigma'. \sum_{\sigma} \rho(\sigma) \times t(\sigma, \sigma')$$

where

$$t(\sigma, \sigma') := \Pr(\sigma, \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else skip}, \sigma')$$

With this definition of f, it is easy to compute f^2 as follows:

$$f^2(\rho) = \lambda\sigma'. \sum_{\sigma} \rho(\sigma) \times t^2(\sigma, \sigma')$$

where $t^2(\sigma, \sigma') := \sum_{\sigma''} t(\sigma, \sigma'')t(\sigma'', \sigma')$ Since the function t maps program states to real numbers, it can be compactly represented as an ADD. As a consequence, t^2 can be efficiently computed and it follows that f^{2^n} is also efficiently computable.

- *Variable ordering*: The size of an ADD crucially depends on the ordering of variables used to construct it. Our compiler implements well-known algorithms [21] using the program's variable dependency graph to determine ADD variable ordering.

- *Batch processing*: Given a joint probability distribution $p(x_1, x_2, \ldots, x_n)$ over n variables, the *marginal distribution* over x_i is the projection of the joint distribution to that variable. Formally, if each of the x_i is a continuous variable ranging over values with a lowerbound ℓ and an upper bound u, the marginal distribution over a particular variable x_i, denoted $p_i(x_i)$ is defined as:

$$\int_{\ell}^{u} \cdots \int_{\ell}^{u} p(x_1, x_2, \ldots, x_n) dx_1 \ldots dx_{i-1} dx_{i+1} \ldots dx_n$$

Computation of marginals for discrete distributions is done by replacing the integrals with discrete summation. With ADDs, such summation is easily implemented by existential quantification:

$$p_i(x_i) = \exists x_1 x_2 \ldots x_{i-1} x_{i+1} \ldots x_n . p(x_1, x_2, \ldots, x_n)$$

When the space required by the ADD for the joint distribution $p(x_1, x_2, \ldots, x_n)$ becomes large, we approximate the distribution using its projection to the tuple of marginal distributions as: $\langle p_1(x_1), p_2(x_2), \ldots, p_n(x_n) \rangle$. The latter representation, though more concise, loses information about correlations between the variables, and is therefore less precise. However, when exact inference runs out of memory, this technique allows us to perform efficient and approximate inference, with a very compact memory representation. We use marginalization to implement a heuristic called *batch processing* as follows. We periodically replace the ADD for the joint distribution by the component marginal distributions for batches of data (for example, every n names in `TrueSkill`, for some value of n), and perform approximate inference.

6. EVALUATION

Benchmarks. We present empirical results from running the inference algorithm on the following benchmarks[1]:

- `Students`: This example is adapted from the advisor-student examples in Markov Logic Networks [17]. We have an array of m students, n teachers, and k courses, and we have information about which teacher is teaching which course, and which student attends which course. The probabilistic program for this example models our belief that if a student s attends a course c taught by teacher t, then we can infer that s likes t (represented by a Bernoulli variable $\mathsf{Likes}(s, t)$) is true with a certain probability p. The goal is to infer the posterior probability of every random variable in the two-dimensional array Likes.

- `Friends`: This benchmark performs probabilistic transitive closure. Given n students and an input friendship matrix (this is an $n \times n$ symmetric matrix F with $F(a, b) = 1$ if student a is a friend of student

[1]The source code for all benchmarks is available in Appendix A.

```
unfairCoin(p) {
  x := p;
  b := true;
  while (b) {
  b := random(Bernoulli 0.5);
  if (b)
    x := 2 * x;
  if (x >= 1.0)
    x := x - 1;
  else if (x >= 0.5)
    x := 1;
  else
    x := 0;
  }
  return x;
}
```

(a)

```
uniform(N) {
  g := N;
  while (g >= N) {
    n := 1;
    g := 0;
    while (n < N) {
      n := 2 * n;
      if (random(Bernoulli(0.5)))
        g := 2 * g;
      else
        g := 2 * g + 1;
    }
  }
  return g;
}
```

(b)

Figure 8: The `unfairCoin` and `uniform` benchmarks.

b, and ? or unknown otherwise), we wish to compute the set of all friends for each student. The probabilistic program in this case encodes a probabilistic transitive closure constraint – that is, the constraint $(F(a,b) = 1) \wedge (F(b,c) = 1) \Rightarrow (F(a,c) = 1)$ holds with a certain probability. The objective is to complete the friends matrix F conditioned on the above constraint.

- `Compare`: We have two n-bit numbers, where each bit is drawn from a Bernoulli distribution. We want to compute posterior probabilities of these distributions conditioned on the observation that the two numbers are unequal.

- `TrueSkill`: This is the TrueSkill model [14] (the simplified version of TrueSkill is shown in Figure 3). We have an array of n players, each of whose skill is drawn from a Gaussian distribution. When player i plays with player j, we observe that the performance of player i in that game (another Gaussian with mean given by the skill of player i), is greater than the performance of player j in that game. Using observations from m such games, we desire to infer the posterior probability distributions for the skills of each player.

- `Loopy programs`: The benchmarks `OneCoin` (Example 3 in Section 1), `Dice`, `unfairCoin` (Figure 8(a)) and `uniform` (Figure 8(b)) are benchmarks that contain loops. The benchmark `Dice`: Figure 8(a) and 8(b) are examples with complex loops [15] that are beyond the scope of existing probabilistic inference solvers. The program `unfairCoin(p)` simulates a biased coin with mean p. Its parameter is the number d of binary digits used in the discretization of real numbers. For the experiments we took an arbitrary value of $p = 0.6$. The program `uniform(N)` simulates a uniform distribution over the interval $[0, N-1]$. In order to demonstrate the generality of our approach, we also consider the benchmark `MC` that is a program representing a Markov chain defined by the following transition matrix:

$$\begin{bmatrix} 0,9 & 0,05 & 0,05 \\ 0,7 & 0 & 0,3 \\ 0,8 & 0 & 0,2 \end{bmatrix}$$

Results. All experiments were performed on an 2.00GHz Intel i7 processor system with 4GB RAM running Microsoft Windows 7. The maximum memory consumed by ADD inference in any of the benchmarks is less than 200MB.

Table 1 compares exact ADD inference with a number of probabilistic inference tools. Each benchmark is associated with a set of parameters that define the size of the problem. The parameters for the `Student`, `Friends` and `Compare` benchmarks are (#students, #courses, #teachers) and (#people), (#width) respectively. We compared our tool with SamIam [9], an inference engine for discrete models (implementing the algorithms Shenoy-Shafer, Hugin, ZC-Hugin and Recursive Conditioning) and OpenBugs an inference engine that employs MCMC sampling. SamIam performs almost as well as the ADD algorithm for the discrete benchmarks. It is important to note that exact ADD inference runs of out memory for the `Friends` benchmark with $p > 6$. This motivates the need for a scalability heuristic like batch processing at the cost of approximation. OpenBugs can quickly give approximate answers on small examples, but is very slow for exact answers. This is due to the fact that with a lot of `observe` statements, it is hard to find valid paths to compute a relevant answer – this is a standard issue with all rejection sampling based techniques. We give the computation times with a number of iterations set to get a posterior probability with a precision of 0.01. Gibbs Sampling (GS) and Expectation Propagation (EP) are inference algorithms available with the Infer.NET [22] toolkit. As seen from the table, ADD is significantly more performant than GS and EP (even though GS and EP compute approximate answers).

Table 2 reports the results for ADD inference with batch processing and discretization. For both the benchmark programs, we use batch processing in order to get an approximate solution whose precision is comparable to the solution obtained by EP (which is promising as EP is used by TrueSkill in Xbox live). For example, in `TrueSkill`, we marginalize the skill variables after every set of n games. For the `TrueSkill` benchmark, ADD also performs discretization in order to handle continuous distributions.

Finally, Table 3 shows the results of ADD inference over the loopy benchmarks. We compare ADD inference with EP. Since EP does not support fixpoints, we unroll the loops in the benchmarks a fixed number of times and then feed the resulting programs to EP. Except for the simple examples, `OneCoin` and `Dice`, the EP algorithm was not able to give the expected distributions. On the other hand, ADD inference converges in a small number of iterations for the loops (generally 2 or 3), thanks to the fast exponentiation method. For the `MC` example, it is important to note that we are essentially computing an ADD representation of the transition matrix, which can be compact in presence of sparse data, and then perform fast exponentiation until convergence (we use KL divergence [8] of distributions across iterations to detect convergence).

7. RELATED WORK

There are a variety of probabilistic programming languages and systems [2, 12, 13, 17, 18, 22, 24]. They perform either dynamic inference either by running the program and performing sampling [5,12], or static inference by first transforming the program to a probabilistic model such as a Bayesian network and then using well known inference algorithms over the transformed model [2,22]. Our technique is static, and in contrast to previous work, perform infer-

Benchmark	Parameters	SHENOY-SHAFER (seconds)	HUGIN (seconds)	ZC-HUGIN (seconds)	REC-COND (seconds)	OPENBUGS (seconds)	GS (seconds)	EP (seconds)	ADD (seconds)
`Students`	s=10, c=10, t=4	0.38	0.40	0.41	0.53	⊥	0.88	1.57	0.11
`Friends`	p=4	0.40	0.41	0.41	0.50	4	7.7	4.09	0.19
	p=5	2.75	2.66	3.37	9.62	18	⊥	12.06	0.42
	p =6	⊥	⊥	⊥	⊥	⊥	⊥	27.3	4.78
`Compare`	n=10	0.28	0.26	0.29	0.33	3	⊥	1.58	0.15
	n=20	0.33	0.31	0.30	0.37	2	⊥	2.34	0.16
	n=100	0.53	0.55	0.52	0.92	6	⊥	12.58	2.15

Table 1: Comparing runtimes of exact ADD inference with other approximate inference algorithms. The ⊥ entry represents a "did not complete".

Benchmark	Parameters	EP (seconds)	ADD (seconds)
`Friends`	p=6	8.89	4.66
	p=7	16.09	10.11
	p =8	30.34	20.4
`TrueSkill`	matches=100, n=1	2.86	2.42
	matches=100, n=2	2.86	2.82
	matches=100, n=3	2.86	3.05
	matches=100, n=4	2.86	4.54
	matches=100, n=5	2.86	5.94
	matches=100, n=7	2.86	9.79
	matches=100, n=10	2.86	25.98
	matches=5000, n=1	33.17	123
	matches=5000, n=2	33.17	153

Table 2: Comparing runtimes of ADD inference (with batch processing and discretization) with Expectation Propagation (EP).

Benchmark	Parameters	EP (seconds)	ADD (seconds)
ONECOIN		0.25	056
DICE		183	0.57
UNFAIRCOIN	d = 5	⊥	0.95
	d = 10	⊥	179
UNIFORM	N = 100	⊥	1.09
	N = 800	⊥	7.54
	N = 2000	⊥	21.50
MC		⊥	0.71

Table 3: Comparing runtimes of ADD inference with Expectation Propagation (EP) for loopy benchmarks.

ence directly over the probabilistic program. Our technique merges data flow facts at join points and hence does not suffer from explosion due to a large number of paths in the program.

Data flow analysis for frequency counting has been explored before by Ramalingam [25]. Geldenhuys et al. [10] use symbolic execution to estimate the probability of executing parts of a program. Both [25] and [10] are frequentist in nature where the probability of a path is obtained via explicit counting. On the other hand, our work is Bayesian in nature where we consider the richer class of probabilistic programs that include sample statements as well as `observe` statements and conditional distributions.

The idea of using ADDs for probabilistic inference has been explored before. Sannar and McAllester [26] define Affine Algebraic Decision Diagrams to perform inference over Bayesian networks and Markov Decision Processes. Kwiatkowska et al. have used a variants of ADDs to perform probabilistic model checking in the PRISM project [20]. Bolzga and Maler have used ADDs to symbolically simulate Markov chains [3]. All these papers study the problem of computing the steady state distribution of Markov chains. Markov chains do not support observe statements, and it is not clear how to encode posterior probability inference efficiently in the framework of Markov chains. Chavira and Darwiche [6] use ADDs to compactly represent factors in a Bayesian network and thereby perform efficient inference via variable elimination. In contrast, we avoid factor graphs altogether and use ADDs to represent symbolic program states (which are distributions) at every program point, much like a data flow analysis or an abstract interpreter [7]. Furthermore, in contrast to graphical models such as Bayesian networks, our technique can handle probabilistic programs with loops.

8. CONCLUSION

We proposed a technique to perform probabilistic inference using data flow analysis. We have also implemented the technique using ADDs as a data structure. We showed that our algorithm indeed computes the posterior probability of a probabilistic program. We have also presented an implementation which shows promising results. We believe this approach opens a door to applying other ideas from program analysis and verification (such as slicing, and abstract interpretation) for doing probabilistic inference.

Acknowledgment. We thank Andreas Podelski and G. Ramalingam for very helpful comments on earlier drafts of this paper.

9. REFERENCES

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[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 (ESOP)*, pages 77–96, 2011. Extended version available as Technical Report MSR–TR–2011–18.

[3] M. Bozga and O. Maler. On the representation of probabilities over structured domains. In *Computer Aided Verification (CAV)*, pages 261–273, 1999.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

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

[6] M. Chavira and A. Darwiche. Compiling bayesian networks using variable elimination. In *International Joint Conference on on Artificial Intelligence (IJCAI)*, pages 2443–2449, 2007.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[8] T. M. Cover and J. A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[9] A. Darwiche. SamIam. Software available from `http://reasoning.cs.ucla.edu/samiam`.

[10] J. Geldenhuys, W. Visser, and M. B. Dwyer. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 166–176, 2012.

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

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

[13] A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. V. Nori, S. K. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *Principles of Programming Languages (POPL)*, pages 403–416, 2013.

[14] R. Herbrich, T. Minka, and T. Graepel. TrueSkill(TM): A Bayesian skill rating system. In *Advances in Neural Information Processing Systems (NIPS)*, pages 569–576, 2007.

[15] J.-P. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs: - automated support for proof-based methods. In *Static Analysis Symposium (SAS)*, pages 390–406, 2010.

[16] G. A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[17] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, University of Washington, 2007. http://alchemy.cs.washington.edu.

[18] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[19] D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

[20] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: a hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[21] S. Malik, A. R. Wang, R. K. Brayton, and A. S. Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design (ICCAD)*, pages 6–9, 1988.

[22] T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from `http://research.microsoft.com/infernet`.

[23] J. Pearl. *Probabilistic reasoning in intelligent systems – networks of plausible inference*. I-XIX. Morgan Kaufmann, 1989.

[24] A. Pfeffer. *Statistical Relational Learning*, chapter The design and implementation of IBAL: A General-Purpose Probabilistic Language. MIT Press, 2007.

[25] G. Ramalingam. Data flow frequency analysis. In *Programming Languages Design and Implementation (PLDI)*, pages 267–277, 1996.

[26] S. Sanner and D. A. McAllester. Affine Algebraic Decision Diagrams (AADDs) and their application to structured probabilistic inference. In *International Joint Conference on on Artificial Intelligence (IJCAI)*, pages 1384–1390, 2005.

[27] F. Somenzi. CUDD: CU decision diagram package, release 2.5.0. Software available from `http://vlsi.colorado.edu`.

APPENDIX

A. BENCHMARK PROGRAMS

```
students(coursesOfStudents, teacherOfCourses,
nStudents, nCourses, nTeachers, student, teacher) {
  for (s=0; s<nStudents; s++)
    for (t=0; t<nTeachers; t++)
      // if a student likes a teacher
      likes[s][t] := random(Bernoulli(0.5));
  // a student may like a teacher if she is
  // attending her classes
  for (s=0; s<nStudents; s++)
    for (c=0; c<nCourses; c++)
      if (random(Bernoulli(0.7)) &&
        coursesOfStudents[s][c])
        observe(likes[s][teacherOfCourses.[c]]);
  return likes[student][teacher];
}
```

Figure 9: The `Students` benchmark.

```
friends(knownFriends, nStudents, student1, student2) {
  // friendship matrix
  for(s1=0; s1<nStudents; s1++)
    for(s2=0; s2<nStudents; s2++)
      friends[s1][s2] = random(Bernoulli(0.5));
  // known friends
  for(s1=0; s1<nStudents; s1++)
    for(s2=0; s2<nStudents; s2++)
      if (knownFriends[s1][s2])
        observe (friends[s1][s2]);
  // friendship is transitive
  for(s1=0; s1<nStudents; s1++)
    for(s2=0; s2<nStudents; s2++)
      for(s3=0; s3<nStudents; s3++)
        if (random(Bernoulli(0.6)) && friends[s1][s2]
          && friends[s2][s3])
          observe(friends[s1][s3]);
  return friends[student1][student2];
}
```

Figure 10: The `Friends` benchmark.

```
compare(n) {
  // binary representation of the integer 'a'
  for(i=0; i<n; i++)
    a[i] = random(Bernoulli(0.5));
  // binary representation of the integer 'b'
  for(i=0; i<n; i++)
    b[i] = random(Bernoulli(0.5));
  // condition 'a <= b'
  r = true;
  for(i=0; i<n; i++)
    r = (a[i] <= b[i]) || (a[i] == b[i] && r);
  return r;
}
```

Figure 11: The `Compare` benchmark.

```
oneCoin() {
  b = false;
  while (not b)
    b = random(Bernoulli(0.5));
  return b;
}
```

Figure 12: The `OneCoin` benchmark.

```
dice() {
  bs = (false, false, false);
  while (bs == (false, false, false) ||
    bs == (true, true, true))
    bs = (random(Bernoulli(0.5)),
          random(Bernoulli(0.5)),
          random(Bernoulli(0.5)));
  return bs;
}
```

Figure 13: The `Dice` benchmark.