# Probabilistic Programming

Andrew D. Gordon
Microsoft Research
adg@microsoft.com

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

Sriram K. Rajamani
Microsoft Research
sriram@microsoft.com

## ABSTRACT

Probabilistic programs are usual functional or imperative programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observations. Models from diverse application areas such as computer vision, coding theory, cryptographic protocols, biology and reliability analysis can be written as probabilistic programs.

Probabilistic inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. Depending on the application, the desired output from inference may vary—we may want to estimate the expected value of some function $f$ with respect to the distribution, or the mode of the distribution, or simply a set of samples drawn from the distribution.

In this paper, we describe connections this research area called "Probabilistic Programming" has with programming languages and software engineering, and this includes language design, and the static and dynamic analysis of programs. We survey current state of the art and speculate on promising directions for future research.

## Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications–Design languages

## General Terms

Languages, Testing, Verification

## Keywords

Machine learning, Program analysis, Probabilistic programming

## 1. INTRODUCTION

*Probabilistic programs* are "usual" programs (written in languages like C, Java, LISP or ML) with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observe statements (which allow data from real world observations to be incorporated into a probabilistic program). A variety of probabilistic programming languages and systems has been proposed [5,19,21,22,31,33,41, 48]. 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. Probabilistic programs can be used to represent *probabilistic graphical models* [32], 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.

*Probabilistic inference* is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. If the probability distribution is over a large number of variables, an explicit representation of the joint probability distribution may be both difficult to obtain efficiently, and unnecessary in the context of specific application contexts. For example, we may want to compute the expected value of some function $f$ with respect to the distribution (which may be more efficient to calculate without representing the entire joint distribution). Alternatively, we may want to calculate the most likely value of the variables, which is the mode of the distribution. Or we may want to simply draw a set of samples from the distribution, to test some other system which expects inputs to follow the modeled distribution.

The goal of probabilistic programming is to enable probabilistic modeling and machine learning to be accessible to the working programmer, who has sufficient domain expertise, but perhaps not enough expertise in probability theory or machine learning. We wish to hide the details of inference inside the compiler and runtime, and enable the programmer to express models using her domain expertise and dramatically increase the number of programmers who can benefit from probabilistic modeling.

The goal of this paper is to give an overview of probabilistic programming for the software engineering community. We assume familiarity with program analysis and program semantics, but provide all the necessary background in probabilistic inference. We use a simple C-like programming language notation for probabilistic programs. We draw connections between probabilistic inference and static and dynamic program analysis. We discuss language design issues,



*FOSE'14*, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2865-4/14/05...$15.00
http://dx.doi.org/10.1145/2593882.2593900

```
1: bool c1, c2;
2: c1 = Bernoulli(0.5);
3: c2 = Bernoulli(0.5);
4: return(c1, c2);
```

1(a)

```
1: bool c1, c2;
2: c1 = Bernoulli(0.5);
3: c2 = Bernoulli(0.5);
4: observe(c1 || c2);
5: return(c1, c2);
```

1(b)

Example 1.

```
1: bool c1, c2;
2: int count = 0;
3: c1 = Bernoulli(0.5);
4: if (c1) then
5:   count = count + 1;
6: c2 = Bernoulli(0.5);
7: if (c2) then
8:   count = count + 1;
9: observe(c1 || c2);
10:return(count);
```

Example 2.

**Figure 1: Simple probabilistic programs.**

and bring out several challenges that remain to be solved in order for probabilistic programming to truly democratize access to machine learning.

In Section 2, we give an introduction to probabilistic programs, starting with examples and ending with a precise formal semantics. In Section 3, we describe the relationship between probabilistic programs and other probabilistic models that readers may be familiar with, such as Markov Chains and Bayesian Networks, and how these models can be encoded as probabilistic programs. In Section 4, we describe how applications from diverse fields such as machine learning, security and biology can be encoded as probabilistic programs. In Section 5, we describe language design issues for probabilistic programs, and how we can use high-level languages such as Excel tables to describe probabilistic programs. In Section 6, we describe techniques for performing probabilistic inference and draw parallels to static and dynamic program analysis. In Section 7, we discuss several open questions and opportunities for future research in probabilistic programming.

## 2. BACKGROUND

We start this section with some examples to familiarize the reader with probabilistic programs, and also informally explain the main ideas behind giving semantics to probabilistic programs. We conclude the section with a precise description of syntax and semantics of probabilistic programs.

**Examples of Simple Probabilistic Programs.** We introduce the syntax and semantics of probabilistic programs using three simple probabilistic programs from Figure 1. The program at the top left, Example 1(a), tosses two fair coins (simulated by draws from a Bernoulli distribution with mean 0.5), and assigns the outcomes of these coin tosses to the Boolean variables `c1` and `c2` respectively, and returns (`c1`, `c2`). The semantics of this program is the expectation of its return value. In this case, this is equal to $(1/2, 1/2)$. Since we have that Pr(`c1`=`false`, `c2`=`false`) = Pr(`c1`=`false`, `c2`=`true`) = Pr(`c1`=`true`, `c2`=`false`) = Pr(`c1`=`true`, `c2`=`true`) = $1/4$, we have that the expectation on the return value is given by $1/4 \times (0,0) + 1/4 \times (0,1) + 1/4 \times (1,0) + 1/4 \times (1,1) = (1/2, 1/2)$ (by treating `true` as 1 and `false` as 0).

The program in Example 1(b) is slightly different from Example 1(a)—it executes the observe statement `observe(c1||c2)` before returning the value of (`c1`, `c2`). The `observe` statement blocks runs which do not satisfy the boolean expression `c1||c2` and does not permit those executions to happen. Executions that satisfy `c1||c2` are permitted to happen. The semantics of the program is the expected return value, conditioned by permitted executions. Since conditioning by permitted executions yields Pr(`c1`=`false`,`c2`=`false`) = 0, and Pr(`c1`=`false`,`c2`=`true`) = Pr(`c1`=`true`,`c2`=`false`) = Pr(`c1`=`true`,`c2`=`true`) = $1/3$, we have that the expected return value is $0\times(0,0)+1/3\times(0,1)+1/3\times(1,0)+1/3\times(1,1) = (2/3, 2/3)$.

```
1: bool c1, c2;
2: int count = 0;
3: c1 = Bernoulli(0.5);
4: if (c1) then
5:   count = count + 1;
6: c2 = Bernoulli(0.5);
7: if (c2) then
8:   count = count + 1;
9: while !(c1 || c2) {
10:  count = 0;
11:  c1 = Bernoulli(0.5);
12:  if (c1) then
13:     count = count + 1;
14:  c2 = Bernoulli(0.5);
15:  if (c2) then
16:     count = count + 1;
17:}
18: return(count);
}
```

Example 3.

```
1: bool b, c;
2: b := 1;
3: c := Bernoulli(0.5);
4: while (c){
5:   b := !b;
6:   c := Bernoulli(0.5);
7: }
8: return(b);
```

Example 4.

**Figure 2: Probabilistic programs with loops.**

Another variation of this example is shown in Example 2. This program also counts the number of coin tosses that result in the value `true`, and stores this count in the variable `count`. The semantics of the program is the expected return value, conditioned by permitted executions. Once again, since conditioning by permitted executions yields Pr(`c1`=`false`,`c2`=`false`) = 0, and Pr(`c1`=`false`,`c2`=`true`) = Pr(`c1`=`true`,`c2`=`false`) = Pr(`c1`=`true`,`c2`=`true`) = $1/3$, we have that the expected return value is $0 \times 0 + 1/3 \times 1 + 1/3 \times 1 + 1/3 \times 2 = 4/3$.

We note that the statement `observe(x)` is very related to the statement `assume(x)` used in program verification literature [2, 15, 42]. Also, we note that `observe(x)` is equivalent to the while-loop `while(!x) skip` since the semantics of probabilistic programs is concerned about the normalized distribution of outputs over terminating runs of the program, and ignores non-terminating runs. However, we use the terminology `observe(x)` because of its common use in probabilistic programming systems [5, 22].

**Loopy Probabilistic Programs.** Figure 2 shows two probabilistic programs with loops. The program in the left side of Figure 2, Example 3, is equivalent to the program in Example 2, in which the observe statement has been equivalently encoded using a while loop. The observe statement in line `9` of Example 2 admits only executions that satisfy the condition `(c1||c2)`. The while loop in lines `9-17` has equivalent functionality to the observe statement. If `(c1||c2)` holds, then the loop exits. If not, it merely re-samples `c1` and `c2`, re-calculates `count` and checks the condition `(c1||c2)` again.

In general, observe statements can be encoded using loops. However, the converse is difficult to do without computing loop invariants. To illustrate this, consider Example 4 on the right side of Figure 2. In this program, the return value of `b` is `1` iff the while loop in lines `3-7` executes an even

| | | | |
|---|---|---|---|
| $x$ | $\in$ | Vars | |
| **uop** | ::= | $\cdots$ | C unary operators |
| **bop** | ::= | $\cdots$ | C binary operators |
| $\varphi, \psi$ | ::= | $\cdots$ | logical formula |
| $\mathcal{E}$ | ::= | | expressions |
| | | $\mid x$ | variable |
| | | $\mid c$ | constant |
| | | $\mid \mathcal{E}_1$ **bop** $\mathcal{E}_2$ | binary operation |
| | | $\mid$ **uop** $\mathcal{E}$ | unary operation |
| $\mathcal{S}$ | ::= | | statements |
| | | $\mid x = \mathcal{E}$ | deterministic assignment |
| | | $\mid x \sim \texttt{Dist}(\bar{\theta})$ | probabilistic assignment |
| | | $\mid$ `skip` | skip |
| | | $\mid$ `observe` $(\varphi)$ | observe |
| | | $\mid \mathcal{S}_1; \mathcal{S}_2$ | sequential composition |
| | | $\mid$ `if` $\mathcal{E}$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$ | conditional composition |
| | | $\mid$ `while` $\mathcal{E}$ `do` $\mathcal{S}$ | while−do loop |
| $\mathcal{P}$ | ::= | $\mathcal{S}$ `return` $(\mathcal{E})$ | program |

**Figure 3: Syntax of Prob.**

- Unnormalized Semantics for Statements

$$[\![\texttt{skip}]\!](f)(\sigma) := f(\sigma)$$
$$[\![x = \mathcal{E}]\!](f)(\sigma) := f(\sigma[x \leftarrow \sigma(\mathcal{E})])$$
$$[\![x \sim \texttt{Dist}(\bar{\theta})]\!](f)(\sigma) := \int_{v \in \text{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\, dv$$
$$[\![\texttt{observe}(\varphi)]\!](f)(\sigma) := \begin{cases} f(\sigma) & \text{if } \sigma(\varphi) = \texttt{true} \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathcal{S}_1; \mathcal{S}_2]\!](f)(\sigma) := [\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f))(\sigma)$$
$$[\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2]\!](f)(\sigma) := \begin{cases} [\![\mathcal{S}_1]\!](f)(\sigma) & \text{if } \sigma(\mathcal{E}) = \texttt{true} \\ [\![\mathcal{S}_2]\!](f)(\sigma) & \text{otherwise} \end{cases}$$
$$[\![\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}]\!](f)(\sigma) := \sup_{n \geq 0} [\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}]\!](f)(\sigma)$$

where

$$\texttt{while}\ \mathcal{E}\ \texttt{do}_0\ \mathcal{S} = \texttt{observe(false)}$$
$$\texttt{while}\ \mathcal{E}\ \texttt{do}_{n+1}\ \mathcal{S} = \texttt{if}\ \mathcal{E}\ \texttt{then}\ (\mathcal{S}; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S})\ \texttt{else}\ (\texttt{skip})$$

- Normalized Semantics for Programs

$$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] := \frac{[\![\mathcal{S}]\!](\lambda\sigma.\ |\sigma(\mathcal{E})|)(\bot)}{[\![\mathcal{S}]\!](\lambda\sigma.\ 1)(\bot)}$$

**Figure 4: Denotational Semantics of Prob: For each statement $\mathcal{S}$, we have $[\![\mathcal{S}]\!] \in (\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$.**

number of times, and it is `0` if the while loop executes an odd number of times. Thus, the expected value of `b` returned by the program, which is the same as the probability that `b` is `1`, is equal to the probability that the while loop executes an even number of times. The probability that the loop executes `0` times is given by 0.5, since this is the same as the probability that `c` is assigned `1` in line `3`. The probability that the loop executes `2` times is equal to the probability that `c` is assigned `0` in line `3`, and that it is assigned `0` the first time line `6` is executed, and that it is assigned `1` the second time line `6` is executed, which is equal to $0.5^3$ since the three assignments to `c` are each independent Bernoulli trials each with probability 0.5. Summing up this for all even number of executions, we get the geometric series $0.5 + 0.5^3 + 0.5^5 + \cdots$ which evaluates to $2/3$.

**Syntax and Semantics.** Now that we have introduced probabilistic programs using simple examples, we proceed to give a precise syntax and semantics. (Our semantics is a variation of one of the classic semantics of probabilistic programs due to Kozen [35].) The probabilistic programming language Prob that we consider is a C-like imperative programming language with two additional statements:

1. The *probabilistic assignment* "$\texttt{x} \sim \texttt{Dist}(\bar{\theta})$" draws a sample from a distribution `Dist` with a vector of parameters $\bar{\theta}$, and assigns it to the variable $x$. For instance, the statement "$\texttt{x} \sim \texttt{Gaussian}(\mu, \sigma)$" draws a value from a Gaussian distribution with mean $\mu$ and standard deviation $\sigma$, and assigns it to the variable $x$.

2. The *observe statement* "$\texttt{observe}(\varphi)$" conditions a distribution with respect to a predicate or condition $\varphi$ that is defined over the variables in the program. In particular, every valid execution of the program must satisfy all conditions in observe statements that occur along the execution.

The syntax of Prob is formally described in Figure 3. A program consists of a statement and a return expression. Variables have base types such as int, bool, float and double. Expressions include variables, constants, binary and unary operations.

Statements include primitive statements (deterministic assignment, probabilistic assignment, observe, skip ) and composite statements (sequential composition, conditionals and loops). Features such as arrays, pointers, structures and function calls can be included in the language, and their treatment does not introduce any additional challenges due to probabilistic semantics. Therefore, we omit these features, and focus on a core language.

The semantics of Prob is described in Figure 4. A state $\sigma$ of a program is a (partial) valuation to all its variables. The set of all states (which can be infinite) is denoted by $\Sigma$. We also consider the natural lifting of $\sigma : \text{Vars} \rightharpoonup \text{Val}$ to expressions $\sigma : \text{Exprs} \to \text{Val}$. We make this lifting a total function by assuming default values for uninitialized variables; we write $\bot$ for the state that assigns the default value to each variable. The definition of the lifting $\sigma$ for constants, unary and binary operations is standard.

The meaning of a probabilistic program is the expected value of its return expression. The return expression of a program is a function $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ from program states to non-negative reals. The denotational semantics $[\![\mathcal{S}]\!](f)(\sigma)$ gives the expected value returned by a program with statement $\mathcal{S}$, return expression $f$, and initial state $\sigma$. The semantics is completely specified using the rules in Figure 4.

The `skip` statement merely applies the return expression $f$ to the input state $\sigma$, since the statement does not change the input state. The deterministic assignment statement first transforms the state $\sigma$ by executing the assignment and then applies $f$. The meaning of the probabilistic assignment is the expected value obtained by sampling $v$ from the distribution `Dist`, executing the assignment with $v$ as the RHS value, and applying $f$ on the resulting state (the expectation is the integral over all possible values $v$). The `observe` statement functions like a `skip` statement if the expression $\varphi$ evaluates to `true` in the initial state $\sigma$, and returns the value 0 otherwise. Due to the presence of observe statements, the semantics of statements shown in Figure 4 is unnormalized. The normalized semantics for programs is obtained by appropriately performing the normalization operation as shown in the second part of Figure 4. The sequential and conditional statements behave as expected and the while-do loop has a standard fixpoint semantics.

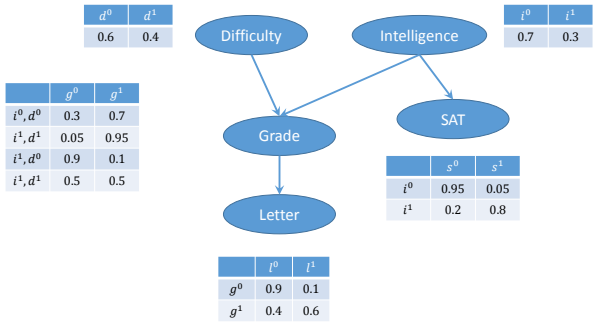


**Figure 5: Bayesian Network example.**

```
1:  bool i, d, s, l, g;
2:  i = Bernoulli(0.3);
3:  d = Bernoulli(0.4);

4:  if (!i && !d)
5:    g = Bernoulli(0.7);
6:  else if (!i && d)
7:    g = Bernoulli(0.95);
8:  else if (i && !d)
9:    g = Bernoulli(0.1);
10: else
11:   g = Bernoulli(0.5);

12: if (!i)
13:   s = Bernoulli(0.05);
14: else
15:   s = Bernoulli(0.8);

16: if (!g)
17:   l = Bernoulli(0.1);
18: else
19:   l = Bernoulli(0.6);

20: return (i,d,g,s,l);
```

(a) Example 5.

```
1:  bool i, d, s, l, g;
2:  i = Bernoulli(0.3);
3:  d = Bernoulli(0.4);

4:  if (!i && !d)
5:    g = Bernoulli(0.7);
6:  else if (!i && d)
7:    g = Bernoulli(0.95);
8:  else if (i && !d)
9:    g = Bernoulli(0.1);
10: else
11:   g = Bernoulli(0.5);
12: observe(g = 1);

13: if (!i)
14:   s = Bernoulli(0.05);
15: else
16:   s = Bernoulli(0.8);

17: if (!g)
18:   l = Bernoulli(0.1);
19: else
20:   l = Bernoulli(0.6);

21: return l;
```

(b) Example 6.

**Figure 6: Encoding Bayesian Networks.**

**From Expectations to Distributions.** Before we conclude this section, we remark that even though we have chosen to give semantics of probabilistic programs as the expected return value of the program, the semantics is fairly general. For instance, if we want to calculate the probability that the program terminates in a particular state $\hat{\sigma}$, we can return a predicate $\sigma = \hat{\sigma}$, which returns 1 iff the state just before returning is $\hat{\sigma}$ and 0 otherwise. The expected value thus returned is PDF of the distribution of output states (also called posterior distribution) evaluated at $\hat{\sigma}$.

# 3. RELATIONSHIPS WITH OTHER MODELS

In this section, we explore the relationships between probabilistic programs and other probabilistic models that readers may have encountered before. In particular, we consider (1) Bayesian Networks and (2) Discrete Time Markov Chains. We show that Bayesian Networks can be encoded as loop-free probabilistic programs. We also show that while Markov Chains can be encoded as loopy probabilistic programs, the steady state probability distribution of Markov Chains can be represented as outputs of probabilistic programs (and hence computed using probabilistic inference) only for a restricted class of Markov Chains, since our semantics for probabilistic programs considers only terminating executions and Markov Chains have non-terminating executions. Finally, we briefly relate probabilistic programs to probabilistic databases and Markov Logic Networks.

## 3.1 Bayesian Networks

A Bayesian Network [32] is a directed acyclic graph $G = \langle V, E \rangle$, where every vertex $v \in V$ is associated with a random variable $X_v$, and every edge $(u, v) \in E$ represents a direct dependence from the random variable $X_u$ to the random variable $X_v$. Let $\mathsf{Deps}(v) = \{u \mid (u, v) \in E\}$ denote the *direct dependences* of node $v \in V$. In a Bayesian Network, each node $v \in V$ of the graph is associated with a *conditional probability distribution* $\mathsf{CPD}(v)$, which denotes the probability distribution of $X_v$ conditioned over the values of the random variables associated with the direct dependences $D(v)$.

Figure 5 shows an example Bayesian Network with 5 nodes and corresponding random variables (1) Difficulty, (2) Intelligence, (3) Grade, (4) SAT and (5) Letter. The example is adapted from [32] and describes a model which relates the intelligence of a student, the difficulty of a course he takes, the grade obtained by the student, the SAT score of the student, and the strength of the recommendation letter obtained by the student from the professor. We denote these random variables with their first letters $I$, $D$, $G$, $S$ and $L$ respectively in the discussion below.

Each of the random variables in this example are discrete random variables and take values from a finite domain. Consequently, the $\mathsf{CPD}$ at each node $v$ can be represented as tables, where rows are indexed by values of the random variables associated with $\mathsf{Deps}(v)$ and the columns are the probabilities associated with each value of $X_v$. For example, the $\mathsf{CPD}$ associated with the node for the random variable $G$ has 2 columns associated with the 2 possible values of $G$ namely $g^0$, $g^1$, and 4 rows corresponding to various combinations of values possible for the direct dependencies of the node namely namely $i^0, d^0$, $i^0, d^1$, $i^1, d^0$, and $i^1, d^1$.

The graph structure together with the $\mathsf{CPD}$ at each node specifies the joint probability distribution over $I$, $D$, $G$, $S$ and $L$ compactly. The probability of a particular state in the joint distribution can be evaluated by starting at the leaf nodes of the Bayesian Network (that is the nodes at the "top" without any inputs) and proceeding in topological order. For instance

$$P(i^1, d^0, g^1, s^1, l^0) =$$
$$P(i^1)P(d^0)P(g^1 \mid i^1, d^0)P(s^1 \mid i^1)P(l^0 \mid g^1) =$$
$$0.3 \times 0.6 \times 0.1 \times 0.8 \times 0.4 = 0.00576$$

Example 5 in the left-side of Figure 6 shows how to encode the Bayesian Network from Figure 5 as a probabilistic program. Note that the program just traverses the Bayesian Network in topological order and performs evaluation according to the $\mathsf{CPD}$ at each node.

Bayesian Networks can be used to pose and answer conditional queries, and this can be encoded in probabilistic

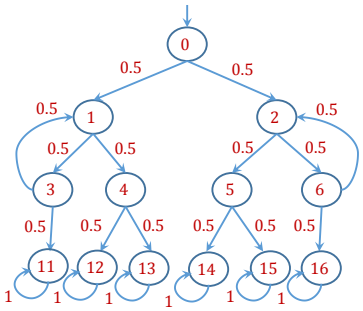


**Figure 7: Discrete Time Markov Chain example.**

```
1:  int x = 0;
2:  while (x < 11) {
3:    bool coin = Bernoulli(0.5);
4:    if(x=0)
5:      if (coin) x = 1 else x = 2;
6:    else if (x=1)
7:      if (coin) x = 3 else x = 4;
8:    else if (x=2)
9:      if (coin) x = 5 else x=  6;
10:   else if (x=3)
11:     if (coin) x = 1 else x = 11;
12:   else if (x=4)
13:     if (coin) x = 12 else x = 13;
14:   else if (x=5)
15:     if (coin) x = 14 else x = 15;
16:   else if (x=6)
17:     if (coin) x = 16 else x = 2;
    }
18: return (x);
```

Example 7.

**Figure 8: Encoding Discrete Time Markov Chains.**

programs using observe statements. For example, we can ask the question $P(L \mid G = g^1)$ which asks for the probability distribution (or the expected value of $L$, given that we observe $G = g^1$). Such a question can be encoded as a probabilistic program shown in Example 6, to the right-side of Figure 6. At line `12` the observe statement `observe(g = 1)` conditions the value of `g` to `1`. Then, at line `21` the program returns `l`. Thus, the meaning of the program is equal to $P(L \mid G = g^1)$.

In general, every Bayesian Network can be encoded as an acyclic probabilistic program in a straightforward manner, and conditioning can be modeled using observe statements in the probabilistic program.

## 3.2 Discrete Time Markov Chains

A Discrete Time Markov Chain (DTMC) is a 4-tuple $\langle V, E, P, v_0 \rangle$, where $V$ is a set of nodes, $E \subseteq V \times V$ is a set of directed edges, $P : E \to \mathbb{R}_{[0,1]}$ labels every edge with a probability such that the sum of the labels of all outgoing edges of a vertex equals 1, and $v_o \in V$ is a distinguished initial vertex.

Informally, we can think of a DTMC $\langle V, E, P, v_0 \rangle$, as starting at $v_0$ and executing an infinite series of steps. At each step, it chooses one of the outgoing edges $e$ of the current vertex with probability $P(e)$ and makes a transition to the target vertex of the edge, and the execution continues. Let $\pi_v(n)$ denote the probability that the chain is in vertex $v$ after $n$ steps of execution. Let $\pi_v$ denote the limit of $\pi_v(n)$ as $n$ tends to infinity. Under certain conditions (such as aperiodicity and ergodicity, see [46]), the limits $\pi_v$ exist for all $v$, and we can think of the vector $\langle \pi_{v_0}, \pi_{v_1}, \ldots \rangle$ as a probability distribution over $V$ (called the steady state distribution of the DTMC).

Figure 7 shows an example DTMC with 13 vertices. The DTMC implements the Knuth-Yao algorithm [30] for obtaining a fair die from fair coin tosses. It turns out that the steady state probability $\pi_v$ for each of the vertices 11 through 16 is 1/6 and for each of the other vertices is 0.

Every DTMC can be encoded as a probabilistic program. Cycles in the DTMC graph can be encoded using while-loops in the probabilistic program. For example, Figure 8 shows a probabilistic program equivalent to the DTMC in in Figure 7.

Unlike the semantics we have given for probabilistic programs, which considers terminating runs (which are finite), the semantics of DTMCs inherently considers infinite runs. However, in cases where every terminal SCC (strongly-connected-component) of a DTMC is a singleton vertex (which is the case, for instance, in the DTMC in Figure 7 where the 6 terminal SCCs are each singleton vertices 11–16), we can terminate the execution of the probabilistic program on reaching the terminal SCC as we show in Figure 8. In cases where the terminal SCC of a DTMC contains multiple vertices, if we want the probabilistic program to compute steady state probabilities of vertices, the semantics in Section 2 will have to be adapted to consider infinite runs.

## 3.3 Extensions

Several extensions to Discrete Time Markov Chains have been widely studied. We briefly mention two such extensions. The first extension is Markov Decision Processes (MDPs) which are obtained by adding nondeterminism to DTMCs. That is, in addition to the probabilistic choice made at each vertex, the model also allows a nondeterministic choice. Resolving the nondeterministic choice at each vertex can be done using a so-called *strategy*, and once a strategy is picked, we obtain a DTMC. Several algorithms for computing strategies in MDPs have been studied [50]. The second extension is Continuous Time Markov Chains (CTMCs), where there is an implicit notion of time, and each transition is annotated with a rate $\gamma$, and in each state the system spends time that is determined by an exponential distribution. Several applications in chemistry (CTMCs are models of well-stirred mixtures of molecules, often biomolecules) and other sciences are naturally expressed as CTMCs. CTMCs can be encoded as probabilistic programs by explicitly introducing time as a program variable, and using a technique known as uniformization [46]. We return to these extensions in later sections.

## 3.4 Probabilistic Databases

A probabilistic database [55] is a relational database in which the tuples stored have associated uncertainties. That

```
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 9: Bayesian skill rating (TrueSkill [26]).**

is, each tuple $t$ has an associated indicator random variable $X_t$ which takes a value 1 if the tuple is present in the database and 0 if the tuple is absent. Each instantiation of values to all of the random variables is called a *world*. The probabilistic database is the joint probability distribution over the random variables, which implicitly specifies a distribution over all the possible worlds. The answer to a query $Q$ on a probabilistic database is the set of tuples $T$ that are possible answers to $Q$, along with a probability $P(t)$ for every $t \in T$ denoting the probability that $t$ belongs the answer for query $Q$. A probabilistic database together with a query can be encoded as a probabilistic program, and the answer to query evaluation can be phrased as probabilistic inference. Work in probabilistic databases has identified a class of queries (called safe queries) for which query evaluation can be performed efficiently, by pushing the probabilistic inference inside particular query plans (called safe plans). Markov Logic Networks [17] use weighted first order logic formulas to construct probabilistic models, and specify distributions over possible worlds. Markov Logic Networks can also be encoded as probabilistic programs. Extensive work has been done on efficient inference algorithms for Markov Logic Networks and these algorithms have been implemented in the tool Alchemy [31].

# 4. APPLICATIONS

In this section, we present applications from various areas including machine learning, clinical diagnosis, ecology and security, and show how these applications can be modeled as probabilistic programs. We use a few language constructs such as function calls and switch statements, which are additions to the syntax presented in Section 2.

**Skill Rating in Online Games.** Online gaming systems such as Microsoft's Xbox Live rate relative skills of players playing online games so as to match players with comparable skills for game playing. The problem is to come up with an estimate of the skill of each player based on the outcome of the games each player has played so far. A Bayesian model for this has been proposed [26]. In Figure 9 we show how this model, called TrueSkill, can be expressed as a probabilistic program. We consider 3 players, A, B and C, whose skills are given by variables `skillA`, `skillB` and `skillC` respectively, which are initialized by drawing

```
int goats, tigers;
double c1, c2, c3, curTime;
// initialize populations
goats = 100; tigers = 4;
// initialize reaction rates
c1 = 1; c2 = 5; c3 = 1;
//initialize time
curTime = 0;

while (curTime < TIMELIMIT)
{
  if (goats > 0 && tigers > 0)
  {
    double rate1, rate2, rate3,
           rate;
    rate1 = c1 * goats;
    rate2 = c2 * goats * tigers;
    rate3 = c3 * tigers;
    rate = rate1 + rate2 + rate3;

    double dwellTime =
              Exponential(rate);
    int discrete =
     Disc3(rate1/rate,rate2/rate);
    curTime += dwellTime;
    switch (discrete)
    {
      case 0: goats++; break;
      case 1: goats--; tigers++;
              break;
      case 2: tigers--; break;
    }
  }
  else if (goats > 0)
  {
    double rate;
    rate = c1 * goats;
    double dwellTime =
         Exponential(rate);
    curTime += dwellTime;
    goats++;
  }
  else if (tigers > 0)
  {
     double rate;
     rate = c3 * tigers;
     double dwellTime =
         Exponential(rate);
     curTime += dwellTime;
     tigers--;
  }
}//end while loop
return(goats,tigers);
}
```

**Figure 10: Lotka-Volterra Population Model.**

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` that are obtained by inference (using the techniques in Section 6) are: `skillA` = `Gaussian`$(105.7, 0.11)$, `skillB` = `Gaussian`$(100.0, 0.11)$, `skillC` = `Gaussian`$(94.3, 0.11)$. Since A won against both B and C, and B won against C, the resulting distribution agrees with our intuitive assessment of their relative skills.

**Predator Prey Population Model.** The Lotka-Volterra predator-prey model [37, 57] is a population model which describes how the population of a predator species and prey

species evolves over time. It is specified using a system of so called "stoichiometric" reactions as follows:

$$\begin{aligned} G &\longrightarrow 2G \\ G+T &\longrightarrow 2T \\ T &\longrightarrow 0 \end{aligned}$$

We consider an ecosystem with goat (denoted by G) and tiger (denoted by T). The first reaction models goat reproduction. The second reaction models consumption of goat by tiger and consequent increase in tiger. The third reaction models death of tiger due to natural causes.

It turns out that this system can be equivalently modeled as a Continuous Time Markov Chain (CTMC) whose state is an ordered pair $(G, T)$ consisting of the number of goats $G$ and the number of tigers $T$. The first reaction can be thought of as a transition in the CTMC from state $(G, T)$ to $(G+1, T)$ and this happens with a rate equal to $c_1 \cdot G$, where $c_1$ is some rate constant, and is enabled only when $G > 0$. Next, the second reaction can be thought of as a transition in the CTMC from state $(G, T)$ to $(G-1, T+1)$ and this happens with a rate equal to $c_2 \cdot G \cdot T$, where $c_2$ is some rate constant, and is enabled only when $G > 0$ and $T > 0$. Finally, the last reaction can be thought of as a transition in the CTMC from from state $(G, T)$ to $(G, T-1)$ and this happens with a rate equal to $c_3 \cdot T$, where $c_3$ is some rate constant, and is enabled only when $T > 0$.

Using a process called uniformization, such a CTMC can be modeled using an embedded DTMC, and encoded as a probabilistic program, shown in Figure 10. The program starts with an initial population of goats and tigers and executes the transitions of the Lotka-Volterra model until a prescribed time limit is reached, and returns the resulting population of goats and tigers. Since the executions are probabilistic, the program models the output distribution of the population of goats and tigers. The body of the while loop is divided into 3 conditions: (1) The first condition models the situation when both goats and tigers exist, and models the situation when all 3 reactions are possible. (2) The second condition models the situation when only goats exist, which is an extreme case, where only reproduction of goats is possible. (3) The third condition models the situation when only tigers exist, which is another extreme case, where only death of tigers is possible.

The encoding of the model as a probabilistic program is complicated, nonmodular and inefficient—at each state, the program has to first find out which reactions are enabled, and then compute total rates of each of the enabled reactions and perform uniformization. A more direct encoding, in terms of both syntax and semantics, is desirable and we discuss this further in Section 7.

**Sensitivity Analysis for Estimating Kidney Disease.** In medical diagnostics, doctors have to often make a decision whether a patient has a particular disease based on measurements from laboratory tests and information from the electronic medical record of patients. However, the test results can be noisy. Further, suppose that there can be transcription errors in the electronic medical records. We wish to evaluate the sensitivity of our decision making because of noisy tests and transcription errors.

Figure 11 shows a probabilistic program that models the system's behavior, taken from [52]. We describe the model, also following assumptions as made in [52]. A common measure of kidney disease in adults is a quantity called *Estimated Glomerular Filtration Rate (eGFR)*, which is computed based on age, gender, ethnicity and measured Serum Creatinine levels of the patient. We use a function `estimateLogEGFR` according to a widely used formula called CKD-EPI [1]. The program considers noise in all the inputs – the Serum Creatinine level (denoted by the `logScr`) variable, the gender (the `isFemale` variable) and ethnicity (the `isAA` variable, which is true if the patient is African American), and calculates the probability that the computed eGFR value can vary by 0.1 or more due to these variations. The variable `bigChange` models the situation when the eGFR value changes by 0.1 or more due to these variations. The program returns this value, and hence can be used to estimate the probability of such an error in eGFR happening due to assumptions on noise in the inputs.

```
double logScr, age;
bool isFemale, isAA;

double f1 =
  estimateLogEGFR(logScr,age,
              isFemale, isAA);
double nLogScr, nAge;
bool   nIsFemale, nisAA;

nLogScr = logScr +
     Uniform(-0.1, 0,1);
nAge = age +
     Uniform(-1, 1);

nIsFemale = isFemale;
if(Bernoulli(0.01))
     nIsFemale = !isFemale;

nIsAA = isAA;
if(Bernoulli(0.01))
     nIsAA = !isAA;

double f2 =
  estimateLogEGFR(nLogScr,nAge,
              nIsFemale, nIsAA);

bool bigChange = 0;
if(f1 - f2 >= 0.1)
    bigChange = 1;
if(f2 - f1 >= 0.1)
    bigChange = 1;

return(bigChange);
```

```
double estimateLogEGFR(
       double logScr, double age,
       bool isFemale, bool isAA)
{
  double k, alpha;
  double f = 4.94;
  if(isFemale){
    k = -0.357;
    alpha= -0.328;
  }
  else{
    k = -0.105;
    alpha = -0.411;
  }

  if(logScr < k)
    f = alpha * (logscr-k);
  else
    f = -1.209 * (logscr-k);

  f = f - 0.007 * age;

  if(isFemale) f = f + 0.017;
  if(isAA) f = f + 0.148;

  return f;
}
```

**Figure 11: Kidney Disease eGFR Sensitivity Estimation.**

**Knowledge-Based Security Policies.** Mardziel and others [39] have explored *knowledge-based security policies*, an access control mechanism based on probabilistic programming. In their work, based on the theory of Clarkson and others [12], a user's agent $U$ receives queries over secret data belonging to the user, and decides whether to accept the query or not by monitoring the knowledge of each querying principal $Q$, represented as a distribution over the secret data. The policy is given by *knowledge thresholds* on aspects of the secrets.

For example, suppose the secret data consists of $U$'s birthday and year, which is say September 27, 1980, and that these are stored in integer variables `bday` $= 270$ and `byear` $=$ 1980. The access control policy is configured in this case by two knowledge thresholds. First, the querier should have less than 20% chance of guessing the birthday. Second, the querier should have less than a 5% chance of guessing both the birthday and the year.

Suppose further that based on demographic information, the prior knowledge of the querier $Q$, an advertiser, is that the values in the ranges $0 \leq$ `bday` $< 365$ and $1956 \leq$ `byear` $<$ 1993 are equally likely. We can represent this prior belief as the following probabilistic program.

[1] http://nephron.com/epi_equation.

```
bday = DiscreteUniform(365);
byear = 1956 + DiscreteUniform(37);
```

Now, suppose the querier wishes to identify those users whose birthday is in the next week to alert them to a special offer. To do so, the advertiser sends the user the following query.

```
output = 0;
today = 260;  // the current date
if ((bday>=today) && (bday<today+7)) output=1;
```

The access control question is whether to allow the advertiser to see the output, and the policy is to allow it so long as the advertiser's knowledge of the secret data after running the query does not exceed the knowledge thresholds stated above.

We decide the access control question by probabilistic inference. First, we infer the posterior distributions of the probabilistic programs obtained from the prior belief plus the query plus either of the two possibilities `observe(output == 0)` or `observe(output == 1)`. Second, in each case we ask whether the posterior probability of any possible value of `bday` exceeds 20%, or whether the probability of any possible values of (`bday`, `byear`) exceeds 5%. If not, it is within policy to return the query result to the advertiser $Q$, but otherwise the query is rejected. After each query from $Q$ is run, we update our representation of the knowledge of $Q$, to be used in the next access control decision.

Notice that access control is decided by checking that the knowledge thresholds would not be exceeded for any possible value of the secret, as opposed to the actual secret. Consider the situation where the advertiser wants to run the query on two consecutive days, first for `today` = 260, and second for `today` = 261. If we use the true secret `bday` = 270 to make the decision, running the query `output` = 0, in both cases, without breaching any of the knowledge thresholds. But suppose in fact the secret was `bday` = 267, then running these two queries would reveal the secret with certainty, as the first would return `output` = 0 and the second `output` = 1, which is only possible if `bday` = 267. Hence, we would need to refuse the second query, after allowing the first, but that refusal itself would leak information about the secret, possibly violating the knowledge thresholds. To avoid this dilemma, the decision is made by quantifying over all possible values of the secrets.

The authors of [39] describe an implementation of their scheme that makes use of a sound abstract interpretation, based on polyhedra, and describe a performance evaluation. This work is still at an early stage, but is a promising avenue for future research.

## 5. LANGUAGE DESIGN

**Imperative, Functional, and Logical Paradigms.** Just as with deterministic programming languages, there are probabilistic languages in the imperative, functional, and logical paradigms. Our prototypical language Prob is imperative, as is the C#-based modeling language underlying Infer.NET [41], and also pWhile, the cryptographic modeling language used by the CertiCrypt [3] and EasyCrypt [4] verification tools for reasoning about cryptographic algorithms and protocols. The most widely adopted probabilistic programming language BUGS [19] is functional. There have been many subsequent functional probabilistic languages including IBAL [48], STAN [54], and Church [21], which is based on the stochastic lambda-calculus. Probabilistic logic languages include BLOG [40], Alchemy [31], and Tuffy [43]. This brief list is far from exhaustive; see the Wiki `probabilistic-programming.org` for discussion of many more probabilistic languages.

**Avoiding the Impedance Mismatch.** The impedance mismatch between programming language types and database schemas is a longstanding problem for conventional programming. For example, programs accessing relational databases need to rely on boilerplate code or libraries to connect external tables to the programming language types such as arrays or collection classes in conventional languages. The same mismatch arises for probabilistic programming languages, with aggregate data being represented in a program as multiple arrays, but in the database as a set of relational tables.

Tabular [23] is a new probabilistic programming language whose most distinctive feature is that programs are written as annotated relational schemas. Conventionally, in SQL or other relational notations, the schema of table describes the types of each item in each row of the table. In Tabular, the conventional schema is enriched with probabilistic expressions that define how to sample each row of the table, hence constituting a probabilistic generative model of the data. The Tabular schema of a table corresponds to the *plate notation* often used to define graphical models.

For example, here is the Tabular schema for the TrueSkill model discussed earlier. There is a table of players and a table of matches between players. We can think of a Tabular schema as a function that given a *concrete database*—our actual data—defines a *predictive database*, a distribution over tables with the same columns as the concrete database but enhanced with *latent columns*. In the case of TrueSkill, the players table has a latent column of skills, and the matches tables has latent columns for the performances.

| Players | | | |
|---|---|---|---|
| Name | string | input | |
| Skill | real | latent | Gaussian(100,10) |
| **Matches** | | | |
| Player1 | link(Players) | input | |
| Player2 | link(Players) | input | |
| Perf1 | real | latent | Gaussian(Player1.Skill,15) |
| Perf2 | real | latent | Gaussian(Player2.Skill,15) |
| Win1 | bool | output | Perf1 > Perf2 |

The schema generates the predictive database as follows. For each row `r` in the concrete Players table, generate:

$$\texttt{Name} = \texttt{r.Name}$$
$$\texttt{Skill} \sim \texttt{Gaussian}(100, 10)$$

For each row `r` in the concrete `Matches` table, generate:

$$\texttt{Player1} = \texttt{r.Player1}$$
$$\texttt{Player2} = \texttt{r.Player2}$$
$$\texttt{Perf1} \sim \texttt{Gaussian}(\texttt{Players}[\texttt{Player1}].\texttt{Skill}, 15)$$
$$\texttt{Perf2} \sim \texttt{Gaussian}(\texttt{Players}[\texttt{Player2}].\texttt{Skill}, 15)$$
$$\texttt{Win1} = (\texttt{Perf1} > \texttt{Perf2})$$
$$\texttt{if}(\texttt{r.Win1!} = \texttt{null})$$
$$\quad \texttt{observe}(\texttt{Win1} == \texttt{r.Win1})$$

The columns marked as *latent* appear only in the predictive database. The *input* and *output* columns must exist in the concrete database. The difference is that there may be `null` entries in output columns, but no missing values are allowed in input columns. The probabilistic expressions on output columns are used to predict missing values, and during the generative process, we add `observe` constraints that

equate the predicted value with the value from the concrete database, when it is not null. Although the model may depend (and hence be conditioned) on input data, it cannot predict missing input values.

Tabular is implemented by translating to an imperative probabilistic program, much like a PROB program, and running Infer.NET to compute posterior marginal distributions for each cell in the predictive database. For a concrete database corresponding to our example in Section 4, where A beats B, B beats C, and A beats C, the result of inference is as follows:

| Players | |
|---|---|
| Name | Skill |
| A | Gaussian(105.7, 0.11) |
| B | Gaussian(100, 0.11) |
| C | Gaussian(94.3, 0.11) |

In a similar way, the predictive form of the Matches table fills in the latent performances.

Early experimental evaluations of Tabular provide some evidence that models are significantly more succinct, and arguably easier to read and understand, than equivalent models written using the Infer.NET input language, while returning the same results without much loss in speed [23]. The hope is that Tabular brings the benefits of probabilistic programming to data enthusiasts who wish to model their data by using a spreadsheet, rather than coding in a full programming environment.

# 6. INFERENCE

Calculating the distribution specified by a probabilistic program is called *probabilistic inference*. The inferred probability distribution is called the *posterior probability* distribution, and the initial guess made by the program is called the *prior probability* distribution. There are a number of methods for performing inference. We briefly note that exact inference is undecidable for programs with unbounded domains. Even for programs with just boolean variables, exact inference is #P-complete [51]. Thus, in this section, we discuss only techniques for approximate inference.

## 6.1 Inference for Bayesian Networks

We first survey inference algorithms for Bayesian networks, For Bayesian networks, efficient and approximate inference is usually performed by message passing algorithms such as *belief propagation* [47] or sampling techniques such as *Markov-Chain-Monte-Carlo (MCMC) algorithms* [38].

**Message Passing.** In Section 2, we saw that a Bayesian network represents a joint probability distribution over its variables. In particular, we also saw that the structure of the Bayesian network represents a particular factorization of the joint probabilistic distribution (using the dependency structure, as a product of the CPD at each node). Message passing algorithms such as belief propagation [47] exploit this factorization in order to efficiently compute the probability distributions for each of the variables in the network.

Consider the Bayesian network shown in Figure 12 due to Pearl [47]. The joint distribution represented by this Bayesian network is:
$P(B, E, J, M) = P(B)P(E)P(A|B, E)P(J|A)P(M|A)$. A query $P(B|J = j^0, M = m^0)$ can be computed as:
$\eta \sum_{E,A} P(B, E, A, J = j^0, M = m^0) =$
$\eta \sum_{E,A} P(B)P(E)P(A|B, E)P(J = j^0|A)P(M = m^0|A)$,

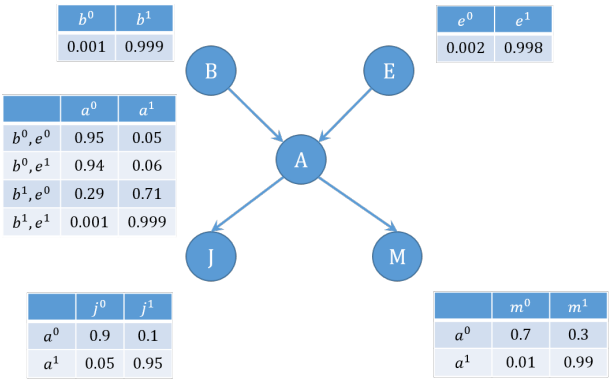


**Figure 12: Bayesian Network for Pearl's example.**

where $\eta$ is a normalization constant. A more efficient computation would exploit the structure of the factorization of $P(B, E, A, J, M)$ that is induced by the structure of the Bayesian network. We note that $P(B|J = j^0, M = m^0)$ can also be written as:
$\eta P(B) \left(\sum_E P(E) \left(\sum_A P(A|B, E)P(J = j^0|A)P(M = m^0|A)\right)\right)$
which involves a fewer number of arithmetic operations than the earlier naive computation. Belief propagation based inference algorithms use such reordering of sum and product operators to compute answers to queries efficiently.

**Sampling.** Sampling based techniques work by drawing samples from the probability distribution represented by the Bayesian network and use the samples to estimate the probability distributions for the variables in the network. Markov Chain Monte Carlo (MCMC) algorithms are widely used for performing such sampling for high-dimensional problems (with a large number of variables). One common MCMC algorithm is the Metropolis-Hastings (MH) algorithm [10]. It makes use of a *proposal distribution* $Q$ to generate samples. In particular, the MH algorithm [10] takes a *target distribution* $P(\bar{x})$ as input (in some implicit form where $P$ can be evaluated at every point) and returns samples that are distributed according to this target distribution. These samples can be used to compute any estimator such as expectation of a function with respect to the target distribution $P(\bar{x})$. Two key steps of the MH algorithm are:

1. Every sample for a variable $x$ is drawn from a proposal distribution $Q(x_{old} \to x_{new})$—this is used to pick a new value $x_{new}$ for the variable $x$ by appropriately perturbing its old value $x_{old}$.

2. A parameter $\beta$ is used to decide whether to accept or reject a new sampled value for $x$, and is defined as follows:

$$\beta = \min\left\{1, \frac{P(x_{new}) \cdot Q(x_{new} \to x_{old})}{P(x_{old}) \cdot Q(x_{old} \to x_{new})}\right\}$$

The sample is accepted if a random value drawn from the uniform distribution over $[0, 1]$ is less than $\beta$, otherwise it is rejected.

It can be shown that repeated application of the above steps will lead to samples which are drawn according to the target distribution $P$ [38].

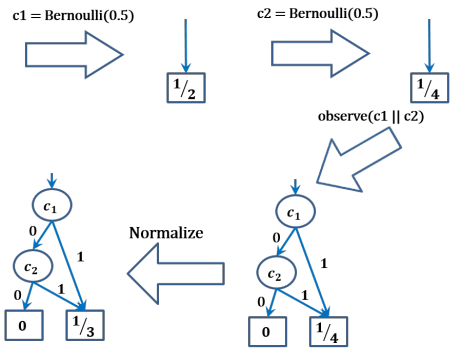


**Figure 13: Data flow analysis of a probabilistic program using ADDs.**

## 6.2 Inference for Probabilistic Programs

A variety of inference techniques have been implemented in probabilistic programming systems [5, 19, 21, 27, 31, 33, 41, 48]. These techniques are variants of the techniques described above for Bayesian networks, and can be broadly classified as follows:

1. **Static inference:** The approach here is to compile the probabilistic program to a probabilistic model such as a Bayesian network, and performing inference using algorithms such as belief propagation and its variants [47]. Infer.NET [41] is one example of such a tool.

2. **Dynamic inference:** Another approach is to execute the program several times using sampling to execute probabilistic statements, observe values of the desired variables in valid runs, and compute statistics on the observed values to infer an approximation to the desired distribution. The Church [21] and Stan [27] probabilistic programming systems are examples of tools that use dynamic inference techniques.

In the remainder of this section, we make important connections between these approaches and work in program analysis. We show that inference *is* program analysis generalized to probabilistic programs [7, 44], and hence explore opportunities for work in the software engineering and program analysis community to contribute to the area of probabilistic programming.

**Static Analysis.** The semantics of a probabilistic program can be calculated using data flow analysis. The data flow facts here are probability distributions and they can be propagated by symbolically executing each statement, merging the data flow facts at join points, and performing fixpoints at loops. In [11] we show how to perform this analysis efficiently using Algebraic Decision Diagrams (ADDs). An ADD [1] is graphical data structure for compactly representing finite functions (in this case a symbolic representation of a joint probability distribution).

We illustrate this analysis for Example 1(b) (from Figure 1) in Figure 13. Symbolically executing the first two statements "`c1 = Bernoulli(0.5)`" and "`c2 = Bernoulli(0.5)`" results in uniform distributions over `c1` and `(c1, c2)` respectively. Both these distributions are compactly represented

```
1:  bool d, i, s, l, g;
2:  d = Bernoulli(0.6);
3:  i = Bernoulli(0.7);

4:  if (!i && !d)
5:    g = Bernoulli(0.3);
6:  else if (!i && d)
7:    g = Bernoulli(0.05);
8:  else if (i && !d)
9:    g = Bernoulli(0.9);
10: else
11:   g = Bernoulli(0.5);

12: if (!i)
13:   s = Bernoulli(0.2);
14: else
15:   s = Bernoulli(0.95);

16: if (!g)
17:   l = Bernoulli(0.1);
18: else
19:   l = Bernoulli(0.4);

20: observe(l = true);
21: return s;
```

(a)

```
1:  bool i, s;
3:  i = Bernoulli(0.7);

12: if (!i)
13:   s = Bernoulli(0.2);
14: else
15:   s = Bernoulli(0.95);

20:  return s;
```

(b)

**Figure 14: Examples to illustrate slicing of probabilistic programs.**

by an Algebraic Decision Diagram (ADD) with a single leaf node. Next, upon processing the statement `observe(c1||c2)`, the analysis computes a sub-distribution represented by the ADD shown in the figure. Finally, upon normalizing this sub-distribution, the final ADD representing the posterior distribution of `(c1, c2)` is obtained. The reader is referred to [11] for further details.

It is also possible to use abstract interpretation [13] based techniques to compute estimates of posterior probabilities—indeed, there is interesting recent work in this direction [39, 52].

Static analysis can also be employed for slicing probabilistic programs [45] and improve the efficiency of inference. Consider the program shown in Figure 14(a). This is a variant of the program shown in Example 6. Usual static program slicing techniques based on control and data program dependences [18] produce the program in Figure 14(b), and this computation is incorrect since the sliced program and the original program can be shown to be not equivalent.

The observe statement (in line 20) introduces new kinds of dependences that are not present in usual programs. Specifically, the observation of the value of `l` influences the value of `g`, which indirectly influences `i`, and ultimately influences `s`. In addition, this flow of influence from `i` to `g` also "opens up" a path for influence to flow from `d` to `i`, and ultimately to `s` as well. This flow of influences is explained using a new notion called *observe dependence* in [45], and is related to *active trails* in Bayesian networks [32]. Observe dependence together with the usual notions of control and data dependence can be used to slice probabilistic programs. Furthermore, the slicing algorithm can be implemented as a source-to-source program transformation, which greatly improves the efficiency of inference by removing irrelevant statements, while providing provably equivalent answers. In the current example, it turns out that *all* the variables `d, i, g, l` and `s` are relevant in this program (and all these dependences are identified if we consider observe dependences together

```
1:  bool earthquake, burglary, alarm, phoneWorking,
    maryWakes,called;
2:  earthquake = Bernoulli(0.001);
3:  burglary = Bernoulli(0.01);
4:  alarm = earthquake || burglary;
5:  if(earthquake)
6:    phoneWorking = Bernoulli(0.6);
7:  else
8:    phoneWorking = Bernoulli(0.99);
9:  if (alarm && earthquake)
10:   maryWakes = Bernoulli(0.8);
11: else if (alarm)
12:   maryWakes = Bernoulli(0.6);
13: else
14:   maryWakes = Bernoulli(0.2);
15: called = maryWakes && phoneWorking;
16: observe(called);
17: return burglary;
```

**Figure 15: Pearl's burglar alarm example.**

with control and data dependences), and the only slice that is semantically equivalent to the program in Figure 14(a) is the entire program!

**Dynamic Analysis.** Dynamic approaches (which are also called sampling based approaches) are widely used, since running a probabilistic program is natural, regardless of the programming language used to express the program. The main challenge in this setting is that many samples that are generated during execution are ultimately rejected for not satisfying the observations. This is analogous to rejection sampling for standard probabilistic models [38]. In order to improve efficiency, it is desirable to avoid generating samples that are later rejected, to the extent possible. In [44], we propose a sampling algorithm R2 that uses program analysis in order to address this challenge. In particular, R2 consists of the following steps:

1. Propagation of observations back through the program using the pre-image operation [16] or Pre analysis to place an observe statement immediately next to every probabilistic assignment. This transformation preserves program semantics (as defined in Figure 4), and helps perform efficient sampling (defined in the next step).

2. Perform a modified Metropolis-Hastings (MH) sampling [10] over the transformed probabilistic program. The modifications exploit the structure in the transformed programs that observe statements immediately follow probabilistic assignments, and sample from subdistributions in order to avoid rejections.

The above two steps prevent rejections due to executions that fail to satisfy observations, and significantly improve the number of accepted MH samples in a given time budget. In contrast, previous approaches [21,49,58,59] have not specifically addressed rejections due to failing observations.

Consider the probabilistic program shown in Figure 15, originally from Pearl's work [47]. This program has probabilistic assignment statements which draw values from distributions. For instance, in line 2, the statement "`earthquake = Bernoulli(0.001)`" draws a value from a Bernoulli distribution with mean `0.001` and assigns it to the variable `x`. The program also has observe statements that are used to condition values of variables. For instance, the statement "`observe(phoneWorking)`" (line 7) blocks all executions

```
1:  bool earthquake, burglary, alarm, phoneWorking,
    maryWakes,called;
2:  earthquake = Bernoulli(0.001);
3:  burglary = Bernoulli(0.01);
4:  alarm = earthquake || burglary;
5:  if(earthquake) {
6:    phoneWorking = Bernoulli(0.6);
7:    observe(phoneWorking);
8:  }
9:  else {
10:   phoneWorking = Bernoulli(0.99);
11:   observe(phoneWorking);
12: }
13: if (alarm && earthquake){
14:   maryWakes = Bernoulli(0.8);
15:   observe(maryWakes && phoneWorking);
16: }
17: else if (alarm){
19:   maryWakes = Bernoulli(0.6);
20:   observe(maryWakes && phoneWorking);
21: }
22: else {
23:   maryWakes = Bernoulli(0.2);
24:   observe(maryWakes && phoneWorking);
25: }
26: called = maryWakes && phoneWorking;
27: return burglary;
```

**Figure 16: Pearl's burglar alarm example after the Pre analysis.**

of the program that do not satisfy the boolean expression (`phoneWorking` = `true`) at line 7. The meaning of a probabilistic program is the expected value of the expression returned by the program with respect to the implicit probability distribution that the program represents. In this example, the variable `burglary` is returned by the program and we are interested in estimating its expected value. Naive sampling, which corresponds to running the program, can result in rejected samples leading to wasteful computation and subsequently loss in efficiency.

For the program shown in Figure 15, R2 first performs a so-called Pre analysis to obtain the transformed program shown in Figure 16. The analysis, which is a backward traversal over the program, starts from the observe statement `observe(called)` in line 16. The value of `called` is calculated using the deterministic assignment, which assigns it the value `maryWakes&&phoneWorking` (line 15). The pre-image of `called` with respect to this assignment statement is `maryWakes&&phoneWorking`, which is propagated back through the program. Next to each probabilistic assignment, the propagated pre-image is inserted as an observe statement. For instance, right after each probabilistic assignment for `maryWakes`, the Pre analysis introduces the observe statement with the predicate `maryWakes&&phoneWorking`. This can be seen in Figure 16, which is the program obtained by applying the Pre analysis to Pearl's burglar alarm example. Also, note that the original observe statement in line 16 of Figure 15 has been deleted in Figure 16. This is because the effect of this observe statement has been propagated backward to other observe statements that are right next to each probabilistic assignment.

The Pre analysis illustrated in the above example is semantics preserving (in other words, the expected values of `burglary` in the programs shown in Figures 15 and 16 are equal) [44]—this is crucial for proving the correctness of R2.

Next, R2 performs sampling over the transformed program in order to compute the expected value of `burglary`.

This is done using a modified version of the MH sampling algorithm that truncates the distribution in each probabilistic assignment statement with the condition in the observe statement following it. More precisely, in the modified MH algorithm, both the proposal distribution and the target density function at each probabilistic assignment statement are modified to use truncated distributions. This ensures that the values drawn from the truncated distributions are such that every program execution is a valid one (that is, the execution satisfies all observe statements), thus avoiding wasteful rejected samples.

We believe that further cross fertilization of ideas from static and dynamic program analysis can improve the precision and scalability of inference for large probabilistic programs, and consider it a promising avenue for future research.

# 7. DISCUSSION

In this section we outline some avenues for future work. We start by discussing the relationship between Probabilistic Programming and Probabilistic Model Checking. While the goals of the communities have been different, we believe that cross-fertilization between the two communities is an interesting direction for future work. We also explore scope for interaction with other sub-communities in machine learning such as the optimization community. We explore adding some new features that are currently absent in probabilistic programming languages—in particular we discuss adding nondeterminism and continuous time (more natively). Finally, we discuss several advances in tool support that are needed for probabilistic programs to be widely usable by non-experts in machine learning.

**Probabilistic Model Checking.** Probabilistic Model Checking involves checking if a probabilistic model satisfies a quantitative specification. Though the idea has been around since the 80's [34, 56], in the past decade there has been a lot of progress in building model checkers such as PRISM [36] and MRMC [29]. PRISM supports various kinds of probabilistic models such as Discrete Time Markov Chains, Continuous Time Markov Chains and Markov Decision Processes. All models are expressed in a guarded-command-like syntax and properties are expressed in probabilistic temporal logics such as PCTL [24]. The properties typically place quantitative bounds on probabilities or expectations. An example property illustrating bounds on probability is "the probability that an airbag fails to deploy within 0.02 seconds after a crash is less than 0.01". Another example which illustrates bounding the expected value is "the expected time for the protocol to finish transmitting one packet is less than 3 milliseconds".

Probabilistic model checking and probabilistic programming systems have evolved as different communities, with different goals, even though the analysis techniques used are very related. The goal of a probabilistic program is to represent (and model) a probability distribution. The view point taken is Bayesian, and typically a probabilistic program assigns to variables from a "prior" distribution, and then constrains the relationships between variables using observe statements, and the goal of the program is to represent a "posterior" distribution obtained by conditioning the prior distribution using observations. The goal of a probabilistic inference (as implemented in a probabilistic programming system) is to compute appropriate representations of this posterior distributions, or expectations of functions with respect to this posterior distribution. In contrast, the goal of probabilistic model checking is to perform verification, that is, to model a system with probabilistic components and verify properties that place quantitative bounds on all possible behaviors of the system.

```
double mean, sd;
mean =
 NDChoice({50,51,52,53,54,55});
variance = 10.0;
result = Gaussian(mean,variance)
return(result);
```

```
double mean, sd;
mean =
 Uniform({50,51,52,53,54,55});
variance = 10,0
result = Gaussian(mean,variance)
return(result);
```

**Figure 17: Illustration of nondeterminism**

However, not withstanding this difference in goals, the two communities still use related techniques. For instance, Multi Terminal Decision Diagrams (MTDDs) are used by PRISM to perform symbolic analysis of finite-state probabilistic models [14], and Algebraic Decision Diagrams (ADDs) have been used by probabilistic programming systems to perform posterior inference [11], as well as by Bayesian Networks to perform inference by variable elimination [9]. Thus, we believe that there is scope for the probabilistic programming and probabilistic model checking communities to build on the learnings of each other.

**Other Approaches in Machine Learning.** Current approaches to Probabilistic Programming are heavily influenced by the Bayesian approach to machine learning. The view of a probabilistic program as a generative model, starting with a prior distribution, and giving semantics to probabilistic programs as posterior distribution after conditioning by observations is distinctly Bayesian in philosophy. It is interesting to explore language notations for expressing machine learning approaches from other sub-communities in machine learning, such as the optimizations community which has built various scalable tools for large scale machine learning (see, for instance [28]). Even with the current semantics of probabilistic programs (which is Bayesian), if our goal is to find the mode of the posterior distribution (for computing maximum likelihood estimate, for instance), then optimization techniques such as gradient descent may be applicable after appropriate smoothing transformations [8] on the probabilistic programs to ensure that the posterior distribution is continuous. This is a promising avenue since optimization techniques scale better than search techniques such as MCMC (which were discussed in Section 6).

**Nondeterminism.** Nondeterminism is a powerful modeling tool to deal with unknown information, as well as to specify abstractions in situations where details are unimportant. For instance, consider the two programs in Figure 17. The program on the left side of Figure 17 models an unknown value being used as mean, where the only information we have is that the mean is one of the values 50, 51, 52, 53, 54 or 55, but we do not know which of these values has been picked. Currently, most probabilistic programming systems are unable to represent (and hence analyze) such a model. Instead, an approximate model based on probabilistic choice, which is shown in the right side of Figure 17 is used. The difference is that in the program on the right side, `mean` is chosen uniformly at random from the

set $\{50, 51, 52, 53, 54, 55\}$. An important difference between the two programs is that the meaning of the program on the left-side is a set of Gaussian distributions (each with a different mean), whereas the meaning of the program in the right-side is a single Gaussian distribution, whose mean is obtained from another distribution.

The use of nondeterminism is also fundamental in Markov Decision Processes, which are widely used in control theory [6] to represent modeling problems for controller synthesis. In such situations, the model has several points where nondeterministic choices are made (and represented as an MDP), and the goal of controller synthesis is to come up with a so called *strategy* which maps states of the system to resolution of the nondeterministic choices the controller should make in order to maximize some objective function (usually specified as some aggregation of rewards obtained through the run of the system). Such controller synthesis algorithms (such as policy iteration or value iteration) are well known in MDP literature and are implemented in controller synthesis tools [6] as well as in Probabilistic Model Checkers such as PRISM [36]. However, the use of nondeterminism as a modeling tool for representing unknown quantities in probabilistic programs is not common. Providing such a facility allows expressing natural models in several situations. However, representing and inferring sets of distributions is more complicated than dealing with a single distribution, and hence there are several technical challenges in adding nondeterminism to probabilistic programs and still be able to provide scalable inference.

**Modeling Continuous Time**. Continuous time models (such as CTMCs) are fundamental model in chemistry, biochemistry, population genetics, and also performance analysis, queuing theory, etc. In CTMCs, the probabilistic choice is not between alternatives, but over time. Events in such models happen at random, typically exponentially distributed in time, and different events race against each other and a transition happens when the first event occurs in a state. While we have shown one way to encode continuous time in probabilistic programs in Section 3 (see Figure 10 for an encoding of the Lotka-Volterra population model), the approach is unsatisfactory for two reasons. First, the approach is non-compositional. If we were to add an extra reaction to the three reactions for the population model in Section 3, the changes to the probabilistic program in Figure 10 are non-local, and we need to change the entire program. This is because, the program in Figure 10 models time as a variable, and computes at each state the set of enabled transitions, and performs case analysis to compute total rates and uniformization at each state. Second, the uniformization approach is known to not work well when there are some reactions with very low rates and some with very high rates. Thus, a more native encoding of real time can be beneficial both for compositionality as well as more efficient, scalable inference. However, adding time complicates the semantics and tooling needed for probabilistic programs and hence needs to be done carefully without exploding the complexity of the entire system.

**Tools and Infrastructure**. The goal of probabilistic programming is to help the large number of programmers who have domain expertise, but lack of expertise in machine learning. Several advances in tools and infrastructure are needed before this goal can become a reality. We outline some directions for such advances here. First, it is interesting to explore learning the structure of probabilistic programs (with some help from the programmer) from the data. Similar ideas have been explored in the context of *structure learning* in Bayesian Networks [25] and Markov Logic Networks [20]. Having the programmer sketch [53] some outline of the program and having the system automatically fill in the details will greatly help improve usability. Next, several improvements are needed in the entire tool chain starting from the compiler, inference system, and runtime infrastructures for probabilistic programs. Compilers need to implement the equivalent of common compiler optimizations for probabilistic programs, and runtimes need to exploit the power of massive parallelism available in today's GPUs and cloud services. Diagnostic information needs to be provided to the programmer to help her identify programming errors, and improve the quality of the programs she writes. Substantial improvements in these areas will need interplay between compiler analysis, machine learning and usability research with programmers and data scientists.

## Acknowledgments

We are grateful to Johannes Borgström, Audris Mockus, Dan Suciu, and Marcin Szymczak for feedback on a draft. This work was supported in part by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling) and the FWF NFN RiSE (Rigorous Systems Engineering).

## 8. 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] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[3] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Principles of Programming Languages (POPL)*, pages 90–101, 2009.

[4] G. Barthe, B. GrÃl'goire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology (CRYPTO)*, pages 71–90. 2011.

[5] 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.

[6] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer Science+Business Media, LLC, 2008.

[7] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics (AISTATS)*, pages 153–160, 2013.

[8] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *Programming Language Design and Implementation*, pages 279–291, 2010.

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

[10] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *American Statistician*, 49(4):327–335, 1995.

[11] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference using data flow analysis. In *Foundations of Software Engineering (FSE)*, pages 92–102, 2013.

[12] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

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

[14] L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 395–410, 2000.

[15] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

[16] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM (CACM)*, 18(8):453–457, 1975.

[17] P. Domingos. Markov logic: a unifying language for knowledge and information management. In *CIKM: ACM Conference on Information and Knowledge Management*, page 519, 2008.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems (TOPLAS).*, 9(3):319–349, 1987.

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

[20] V. Gogate, W. A. Webb, and P. Domingos. Learning efficient markov networks. In *Neural Information Processing Systems (NIPS)*, pages 748–756, 2010.

[21] 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.

[22] 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.

[23] A. D. Gordon, T. Graepel, N. Rolland, C. Russo, J. Borgström, and J. Guiver. Tabular: A schema-driven probabilistic programming language. In *Principles of Programming Languages (POPL)*, 2014. To appear.

[24] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[25] D. Heckerman, D. Geiger, and D. M. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[26] R. Herbrich, T. Minka, and T. Graepel. TrueSkill: A Bayesian skill rating system. In *Neural Information Processing Systems (NIPS)*, pages 569–576, 2006.

[27] M. D. Hoffman and A. Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.

[28] John Langford. Vowpal Wabbit.

[29] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. In *Quantitative Evaluation of Systems (QEST)*, pages 167–176, 2009.

[30] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[31] 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.

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

[33] 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.

[34] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[35] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[36] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV)*, pages 585–591, 2011.

[37] A. Lotka. *Elements of physical biology*. Williams & Wilkins company, Baltimore, 1925.

[38] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[39] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, January 2013.

[40] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Probabilistic, Logical and Relational Learning — A Further Synthesis*, 2005.

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

[42] G. Nelson. A generalization of Dijkstra's calculus. *Transactions on Programming Languages and Systems (TOPLAS).*, 11(4):517–561, 1989.

[43] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy:

Scaling up statistical inference in Markov Logic Networks using an RDBMS. *Very Large Databases (PVLDB)*, 4(6):373–384, 2011.

[44] A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel. The R2 probabilistic programming system. 2013. `http://research.microsoft.com/r2`.

[45] A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. 2013. Working draft.

[46] J. R. Norris. *Markov chains*. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998.

[47] J. Pearl. *Probabilistic Reasoning in Intelligence Systems*. Morgan Kaufmann, 1996.

[48] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.

[49] A. Pfeffer. A general importance sampling algorithm for probabilistic programs. Technical report, Harvard University TR-12-07, 2007.

[50] M. L. Puterman. *Markov Decision Processes*. Wiley, 1994.

[51] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[52] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis of probabilistic programs: Inferring whole program properties from finitely many executions. In *Programming Languages Design and Implementation (PLDI)*, 2013.

[53] A. Solar-Lezama. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 4–13, 2009.

[54] Stan Development Team. Stan: A C++ library for probability and sampling, version 2.0, 2013.

[55] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[56] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Foundations of Computer Science (FOCS)*, pages 327–338, 1985.

[57] V. Volterra. Fluctuations in the abundance of a species considered mathematically. *Nature*, 118:558–560, 1926.

[58] D. Wingate, N. D. Goodman, A. Stuhlmüller, and J. M. Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *Neural Information Processing Systems (NIPS)*, pages 1152–1160, 2011.

[59] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 15:770–778, 2011.