TLP **11** (4–5): 663–680, 2011. © Cambridge University Press 2011
doi:10.1017/S1471068411000238

The magic of logical inference in probabilistic programming

BERND GUTMANN, INGO THON, ANGELIKA KIMMIG,
MAURICE BRUYNOOGHE and LUC DE RAEDT

Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A-bus 2402, 3001 Heverlee, Belgium

(*e-mail:* {bernd.gutmann,ingo.thon,angelika.kimmig,
maurice.bruynooghe,luc.deraedt}@cs.kuleuven.be)

Abstract

Today, there exist many different probabilistic programming languages as well as more inference mechanisms for these languages. Still, most logic programming-based languages use backward reasoning based on Selective Linear Definite resolution for inference. While these methods are typically computationally efficient, they often can neither handle infinite and/or continuous distributions nor evidence. To overcome these limitations, we introduce distributional clauses, a variation and extension of Sato's distribution semantics. We also contribute a novel approximate inference method that integrates forward reasoning with importance sampling, a well-known technique for probabilistic inference. In order to achieve efficiency, we integrate two logic programming techniques to direct forward sampling. Magic sets are used to focus on relevant parts of the program, while the integration of backward reasoning allows one to identify and avoid regions of the sample space that are inconsistent with the evidence.

KEYWORDS: probabilistic logic, forward reasoning, sampling, continuous distributions

1 Introduction

The advent of statistical relational learning (De Raedt *et al.* 2008; Getoor and Taskar 2007) and probabilistic programming (De Raedt *et al.* 2008) has resulted in a vast number of different languages and systems such as PRISM (Sato and Kameya 2001), ICL (Poole 2008), ProbLog (De Raedt *et al.* 2007), Dyna (Eisner *et al.* 2005), BLPs (Kersting and De Raedt 2008), CLP($\mathcal{BN}$) (Santos Costa *et al.* 2008), BLOG (Milch *et al.* 2005), Church (Goodman *et al.* 2008), IBAL (Pfeffer 2001), and MLNs (Richardson and Domingos 2006). While inference in these languages generally involves evaluating the probability distribution defined by the model, often conditioned on evidence in the form of known truth values for some atoms, this diversity of systems has led to a variety of inference approaches. Languages such as IBAL, BLPs, MLNs, and CLP($\mathcal{BN}$) combine knowledge-based model construction to generate a graphical model with standard inference techniques for such models. Some probabilistic programming languages, for instance, BLOG

and Church, use sampling for approximate inference in generative models, that is, they estimate probabilities from a large number of randomly generated program traces. Finally, probabilistic logic programming frameworks, such as ICL, PRISM, and ProbLog, combine Selective Linear Definite (SLD) resolution with probability calculations.

So far, the second approach based on sampling has received little attention in logic programming-based systems. In this paper, we investigate the integration of sampling-based approaches into probabilistic logic programming frameworks to broaden the applicability of these programming languages. Particularly relevant in this regard are the ability of Church and BLOG to sample from continuous distributions and to answer conditional queries of the form $p(q|e)$, where e is the evidence. In order to accommodate (continuous and discrete) distributions, we introduce *distributional clauses*, which define random variables together with their associated distributions, conditional upon logical predicates. Random variables can be passed around in the logic program and the outcome of a random variable can be compared with other values by means of special built-ins. In order to formally establish the semantics of this new construct, we show that these random variables define a basic distribution over facts (using the comparison built-ins) as required in Sato's (1995) distribution semantics, and thus induces a distribution over the least Herbrand models of the program. This contrasts with previous instances of the distribution semantics, as we no longer enumerate the probabilities of alternatives but instead use arbitrary densities and distributions.

From a logic programming perspective, BLOG (Milch *et al.* 2005) and related approaches perform *forward reasoning*, that is, the samples needed for probability estimation are generated starting from known facts and deriving additional facts, thus generating a *possible world*. PRISM and related approaches follow the opposite approach of *backward reasoning*, where inference starts from a query and follows a chain of rules backwards to the basic facts, thus generating *proofs*. This difference is one of the reasons for using sampling in the first approach: Exact forward inference would require that all possible worlds be generated, which is infeasible in most of the cases. Based on this observation, we contribute a new inference method for probabilistic logic programming that combines sampling-based inference techniques with forward reasoning. On the probabilistic side, the approach uses rejection sampling (Koller and Friedman 2009), a well-known sampling technique that rejects samples that are inconsistent with the evidence. On the logic programming side, we adapt the *magic set* technique (Bancilhon et al. 1986) toward the probabilistic setting, thereby combining the advantages of forward and backward reasoning. Furthermore, the inference algorithm is improved along the lines of the *SampleSearch* algorithm (Gogate and Dechter 2011), which avoids choices leading to a sample that cannot be used in the probability estimation due to inconsistency with the evidence. We realize this using a heuristic based on backward reasoning with limited proof length, the benefit of which is experimentally confirmed. This novel approach to inference creates a number of new possibilities for applications of probabilistic logic programming systems, including continuous distributions and Bayesian inference.

This paper is organized as follows. We start by reviewing the basic concepts in Section 2. Section 3 introduces the new language and its semantics. Section 4 contains a novel forward sampling algorithm for probabilistic logic programs. Before concluding, we evaluate our approach in Section 5.

2 Preliminaries

2.1 Probabilistic inference

A discrete probabilistic model defines a probability distribution $p(\cdot)$ over a set Ω of basic outcomes, that is, value assignments to the model's random variables. This distribution can then be used to evaluate a conditional probability distribution $p(q|e) = \frac{p(q \wedge e)}{p(e)}$, also called *target distribution*. Here, q is a query involving random variables, and e is the *evidence*, that is, a partial value assignment of the random variables.[1] Evaluating this target distribution is called *probabilistic inference* (Koller and Friedman 2009) In probabilistic logic programming, random variables often correspond to ground atoms, and $p(\cdot)$ thus defines a distribution over truth value assignments, as we will see in more detail in Section 2.3 (but see also De Raedt *et al.* 2008). Probabilistic inference then asks for the probability of a logical query being true given truth value assignments for a number of such ground atoms.

In general, the probability $p(\cdot)$ of query q is in the discrete case the sum over those outcomes $\omega \in \Omega$ that are consistent with the query. In the continuous case, the sum is replaced by an (multidimensional) integral and the distribution $p(\cdot)$ by a (product of) densities $\mathbf{F}(\cdot)$ That is,

$$p(q) = \sum_{\omega \in \Omega} p(\omega) \mathbb{1}_q(\omega), \qquad \text{and} \qquad p(q) = \int_{\Omega} \cdots \int \mathbb{1}_q(\omega) d\mathbf{F}(\omega), \tag{1}$$

where $\mathbb{1}_q(\omega) = 1$ if $\omega \models q$ and 0 otherwise. As common (e.g., Wasserman 2003) we will use for convenience the notation $\int x dF(x)$ as unifying notation for both discrete and continuous distributions.

As Ω is often very large or even infinite, exact inference based on the summation in (1) quickly becomes infeasible, and inference has to resort to approximation techniques based on *samples*, that is, randomly drawn outcomes $\omega \in \Omega$. Given a large set of such samples $\{s_1, \ldots, s_N\}$ drawn from $p(\cdot)$, the probability $p(q)$ can be estimated as the fraction of samples where q is true. Instead, if samples are drawn from the target distribution $p(\cdot|e)$, the latter can directly be estimated as

$$\hat{p}(q|e) := \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_q(s_i).$$

However, sampling from $p(\cdot|e)$ is often highly inefficient or infeasible in practice, as the evidence needs to be taken into account. For instance, if one would use the standard definition of conditional probability to generate samples from $p(\cdot)$, all

[1] If e contains assignments to continuous variables, then $p(e)$ is zero. Hence, evidence on continuous values has to be defined via a probability density function, also called sensor model.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

samples that are not consistent with the evidence do not contribute to the estimate and would thus have to be discarded or, in sampling terminology, *rejected*.

More advanced sampling methods therefore often resort to the so-called *proposal distribution*, which allows for easier sampling. The error introduced by this simplification then needs to be accounted for when generating the estimate from the set of samples. An example for such a method is *importance sampling*, where each sample s_i has an associated *weight* w_i. Samples are drawn from an *importance distribution* $\pi(\cdot|e)$, and weights are defined as $w_i = \frac{p(s_i|e)}{\pi(s_i|e)}$. The true target distribution can then be estimated as

$$\hat{p}(q|e) = \frac{1}{W} \sum_{i=1}^{N} w_i \cdot \mathbb{1}_q(s_i),$$

where $W = \sum_i w_i$ is a normalization constant. The simplest instance of this algorithm is *rejection sampling* as already sketched above, where the samples are drawn from the prior distribution $p(\cdot)$ and weights are 1 for those samples consistent with the evidence, and 0 for others. Especially for evidence with low probability, rejection sampling suffers from a very high rejection rate, that is, many samples are generated but do not contribute to the final estimate. This is known as the *rejection problem*. One way to address this problem is *likelihood weighted sampling*, which dynamically adapts the proposal distribution during sampling to avoid choosing values for random variables that cause the sample to become inconsistent with the evidence. Again, this requires corresponding modifications of the associated weights in order to produce correct estimates.

2.2 Logical inference

A (definite) clause is an expression of the form $\mathtt{h}$:- $\mathtt{b}_1, \ldots, \mathtt{b}_n$, where $\mathtt{h}$ is called head and $\mathtt{b}_1, \ldots, \mathtt{b}_n$ is the body. A program consists of a set of clauses and its semantics is given by its least Herbrand model. There are at least two ways of using a definite clause in a logical derivation. First, there is *backward chaining*, which states that to prove a goal $\mathtt{h}$ with the clause it suffices to prove $\mathtt{b}_1, \ldots, \mathtt{b}_n$; second, there is *forward chaining*, which starts from a set of known facts $\mathtt{b}_1, \ldots, \mathtt{b}_n$ and the clause and concludes that $\mathtt{h}$ also holds (cf. Nilsson and Małiszyński 1996). Prolog employs backward chaining (SLD-resolution) to answer queries. SLD-resolution is very efficient both in terms of time and space. However, similar subgoals may be derived for multiple times if the query contains recursive calls. Moreover, SLD-resolution is not guaranteed to always terminate (when searching depth-first). Using forward reasoning, one starts with what is known and employs the immediate consequence operator T_P until a fixpoint is reached. This fixpoint is identical to the least Herbrand model.

Definition 1 (T_P *operator*)
Let P be a logic program containing a set of definite clauses and $ground(P)$ is the set of all ground instances of these clauses. Starting from a set of ground facts S,

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

the T_P operator returns

$$T_P(S) = \{\mathtt{h} \mid \mathtt{h} \text{ :- } \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathit{ground}(P) \text{ and } \{\mathtt{b_1}, \ldots, \mathtt{b_n}\} \subseteq S\}.$$

2.3 Distribution semantics

Sato's (1995) distribution semantics extends logic programming to the probabilistic setting by randomly choosing truth values of basic facts. The core of this semantics lies in splitting the logic program into a set F of *facts* and a set R of *rules*. Given a probability distribution P_F over the facts, the rules then allow one to extend P_F into a distribution over the least Herbrand models of logic program. Such a Herbrand model is called a *possible world*.

More precisely, it is assumed that $DB = F \cup R$ is ground and denumerable, and that no atom in F unifies with the head of a rule in R. Each truth value assignment to F gives rise to a unique least Herbrand model of DB. Thus, a probability distribution P_F over F can directly be extended into a distribution P_{DB} over these models. Furthermore, Sato (1995) shows that, given an enumeration $f_1, f_2, \ldots$ of facts in F, P_F can be constructed from a series of finite distributions $P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n)$ provided that the series fulfills the so-called compatibility condition, that is,

$$P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n) = \sum_{x_{n+1}} P_F^{(n+1)}(f_1 = x_1, \ldots, f_{n+1} = x_{n+1}). \tag{2}$$

3 Syntax and semantics

Sato's (1995) distribution semantics, as summarized in Section 2.3, provides the basis for most of the probabilistic logic programming languages, including PRISM (Sato and Kameya 2001), ICL (Poole 2008), CP-logic (Vennekens *et al.* 2009), and ProbLog (De Raedt *et al.* 2007). The precise way of defining the basic distribution P_F differs among languages, though the theoretical foundations are essentially the same. The most basic instance of the distribution semantics, employed by ProbLog, uses the so-called *probabilistic facts*. Each ground instance of the *probabilistic fact* directly corresponds to an independent random variable that takes either the value "true" or "false." These probabilistic facts can also be seen as binary switches (cf. Sato 1995), which again can be extended to multi-ary switches or choices as used by PRISM and ICL. For switches, at most one of the probabilistic facts belonging to the switch is "true" according to the specified distribution. Finally, in CP-logic, such choices are used in the head of rules leading to the so-called *annotated disjunction*.

Hybrid ProbLog (Gutmann *et al.* 2010) extends the distribution semantics with continuous distributions. In order to allow for exact inference, Hybrid ProbLog imposes severe restrictions on the distributions and their further use in the program. Two sampled values, for instance, cannot be compared against each other. Only comparisons that involve one sampled value and one number constant are allowed. Sampled values may not be used in arithmetic expressions or as parameters for other distributions; for instance, it is not possible to sample a value and use it as

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

the mean of the Gaussian distribution. It is also not possible to reason over an unknown number of objects as BLOG (Milch *et al.* 2005) does, though this case is mainly for algorithmic reasons.

Here we alleviate these restrictions by defining the basic distribution P_F over probabilistic facts based on both discrete and continuous random variables. We use a three-step approach to define this distribution. First, we introduce explicit random variables and corresponding distributions over their domains, both denoted by terms. Second, we use a mapping from these terms to terms denoting (sampled) outcomes, which then are used to define the basic distribution P_F on the level of probabilistic facts. For instance, assume that an urn contains an unknown number of balls, where the number is drawn from a Poisson distribution and we say that this urn contains many balls if it contains at least 10 balls. We introduce a random variable `number`, and define `many :- dist_gt(≃(number), 9)`. Here, `≃(number)` is the Herbrand term denoting the sampled value of `number`, and `dist_gt(≃(number), 9)` is a probabilistic fact whose probability of being true is the expectation that this value is actually greater than 9. This probability then carries over to the derived atom `many` as well. We will elaborate on the details in the following.

3.1 Syntax

In a logic program, following Sato (1995), we distinguish between probabilistic facts, which are used to define the basic distribution, and rules, which are used to derive additional atoms.[2] Probabilistic facts are not allowed to unify with any rule head. The distribution over facts is based on random variables whose distributions we define through the so-called distributional clauses.

Definition 2 (*Distributional clause*)
A *distributional clause* is a definite clause with an atom $\mathtt{h} \sim \mathscr{D}$ in the head, where $\sim$ is a binary predicate used in infix notation.

For each ground instance $(\mathtt{h} \sim \mathscr{D} \text{ :- } \mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$ with θ being a substitution over the Herbrand universe of the logic program, a distributional clause defines a random variable $\mathtt{h}\theta$ and an associated distribution $\mathscr{D}\theta$. In fact, the distribution is only defined when $(\mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$ is true in the semantics of the logic program. These random variables are terms of the Herbrand universe and can be used as any other term in the logic program. Furthermore, the term $\simeq(d)$ constructed from the reserved functor $\simeq/1$ represents the outcome of the random variable d. These functors can be used inside calls to special predicates in $dist\_rel = \{dist\_eq/2, dist\_lt/2, dist\_leq/2, dist\_gt/2, dist\_geq/2\}$. We assume that there is a fact for each of the ground instances of these predicate calls. These facts are the *probabilistic facts* of Sato's (1995) distribution semantics. Note that the set of probabilistic facts is enumerable as the Herbrand universe of the program is enumerable. The term $\simeq(d)$ links the random variable d with its outcome. The probabilistic facts compare the outcome of a random variable with a constant

[2] A rule can have an empty body, in which case it represents a deterministic fact.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

or the outcome of another random variable and succeed or fail according to the probability distribution(s) of the random variable(s).

Example 1 (*Distributional clauses*)

$$\texttt{nballs} \sim \texttt{poisson}(6). \tag{3}$$

$$\texttt{color}(\texttt{B}) \sim \ [0.7 : \texttt{b}, 0.3 : \texttt{g}] \ \texttt{:-}\ \texttt{between}(1, \simeq(\texttt{nballs}), \texttt{B}). \tag{4}$$

$$\texttt{diameter}(\texttt{B}, \texttt{MD}) \sim \texttt{gamma}(\texttt{MD}/20, 20) \ \texttt{:-}\ \texttt{between}(1, \simeq(\texttt{nballs}), \texttt{B}), \texttt{mean\_diameter}(\simeq(\texttt{color}(\texttt{B})), \texttt{MD}). \tag{5}$$

The defined distributions depend on the following logical clauses:

$$\texttt{mean\_diameter}(\texttt{C}, 5) \ \texttt{:-}\ \texttt{dist\_eq}(\texttt{C}, \texttt{b}).$$

$$\texttt{mean\_diameter}(\texttt{C}, 10) \ \texttt{:-}\ \texttt{dist\_eq}(\texttt{C}, \texttt{g}).$$

$$\texttt{between}(\texttt{I}, \texttt{J}, \texttt{I}) \ \texttt{:-}\ \texttt{dist\_leq}(\texttt{I}, \texttt{J}).$$

$$\texttt{between}(\texttt{I}, \texttt{J}, \texttt{K}) \ \texttt{:-}\ \texttt{dist\_lt}(\texttt{I}, \texttt{J}),\ \texttt{I1 is I} + 1, \texttt{between}(\texttt{I1}, \texttt{J}, \texttt{K}).$$

The distributional clause (3) models the number of balls as a Poisson distribution with mean 6. The distributional clause (4) models a discrete distribution for the random variable color(B). With probability 0.7 the ball is blue and green otherwise. Note that the distribution is defined only for the values B for which $between(1, \simeq(nballs), B)$ succeeds. The execution of calls to the latter gives rise to calls to probabilistic facts that are instances of $dist\_leq(I, \simeq(nballs))$ and $dist\_lt(I, \simeq(nballs))$. Similarly, the distributional clause (5) defines a gamma distribution that is also defined conditionally. Note that the conditions in the distribution depend on calls of the form $mean\_diameter(\simeq(color(n)), MD)$ with n being a value returned by between/3. Execution of this call finally leads to calls $dist\_eq(\simeq(color(n)), b)$ and $dist\_eq(\simeq(color(n)), g)$.

It looks feasible to allow $\simeq(d)$ terms everywhere and to have a simple program analysis insert special predicates at appropriate places by replacing $< /2$, $> /2$, $\leqslant /2$, $\geqslant /2$ predicates by $dist\_rel/2$ facts. Though extending unification is a bit harder, as long as a $\simeq(h)$ term is unified with a free variable, standard unification can be performed only when the other term is bound and extension is required. In this paper we assume that the special predicates dist_eq/2, dist_lt/2, dist_leq/2, dist_gt/2, and dist_geq/2 are used whenever the outcome of a random variable needs to be compared with another value and that it is safe to use standard unification whenever a $\simeq$(h) term is used in another predicate.

For the basic distribution on facts to be well defined, a program has to fulfill a set of validity criteria that has to be enforced by the programmer.

Definition 3 (*Valid program*)
A program P is called *valid* if following conditions are fulfilled:

(V1) In the relation $\texttt{h} \sim \mathscr{D}$ that holds in the least fixpoint of a program, there is a functional dependency from h to $\mathscr{D}$, so there is a unique ground distribution $\mathscr{D}$ for each ground random variable h.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

(V2) The program is *distribution-stratified*, that is, there exists a function $rank(\cdot)$ that maps ground atoms to $\mathbb{N}$ and satisfies the following properties: (1) for each ground instance of a distribution clause $\mathtt{h} \sim \mathscr{D}$:- $\mathtt{b}_1, \ldots \mathtt{b}_n$ holds $rank(\mathtt{h} \sim \mathscr{D} > rank(\mathtt{b}_i)$ (for all i). (2) for each ground instance of another program clause $\mathtt{h}$:- $\mathtt{b}_1, \ldots \mathtt{b}_n$ holds $rank(\mathtt{h}) \geqslant rank(\mathtt{b}_i)$ (for all i). (3) for each ground atom $\mathtt{b}$ that contains (the name of) a random variable $\mathtt{h}$, $rank(\mathtt{b}) \geqslant rank(\mathtt{h} \sim \mathscr{D})$ (with $\mathtt{h} \sim \mathscr{D}$ being the head of the distribution clause defining $\mathtt{h}$).

(V3) All ground probabilistic facts or, to be more precise, the corresponding indicator functions are *Lebesgue-measurable*.

(V4) Each atom in the least fixpoint can be derived from a finite number of probabilistic facts (*finite support condition* (Sato 1995)).

Together, (V1) and (V2) ensure that a single basic distribution P_F over probabilistic facts can be obtained from the distributions of individual random variables defined in P. The (V3) requirement is crucial. It ensures that the series of distributions $P_F^{(n)}$ needed to construct this basic distribution is well defined. Finally, the number of facts over which the basic distribution is defined needs to be countable. This is true, as we have a finite number of constants and functors: those appearing in the program.

3.2 Distribution semantics

We now define the series of distributions $P_F^{(n)}$, where we fix an enumeration $f_1, f_2, \ldots$ of probabilistic facts such that $i < j \implies rank(f_i) \leqslant rank(f_j)$ where $rank(\cdot)$ is a *ranking function* showing that the program is distribution stratified. For each predicate $\mathtt{rel/2} \in dist\_rel$, we define an *indicator function* as follows:

$$I^1_{rel}(X_1, X_2) = \begin{cases} 1 & \text{if } \mathtt{rel}(X_1, X_2) \text{ is true,} \\ 0 & \text{if } \mathtt{rel}(X_1, X_2) \text{ is false.} \end{cases} \tag{6}$$

Furthermore, we set $I^0_{rel}(X_1, X_2) = 1.0 - I^1_{rel}(X_1, X_2)$. We then use the expected value of the indicator function to define probability distributions $P_F^{(n)}$ over finite sets of ground facts $f_1, \ldots, f_n$. Let $\{rv_1, \ldots rv_m\}$ be the set of random variables these n facts depend upon, ordered such that if $rank(rv_i) < rank(rv_j)$, then $i < j$ (cf. (V2) in Definition 3). Furthermore, let $f_i = rel_i(t_{i1}, t_{i2})$, $x_j \in \{1, 0\}$, and $\theta^{-1} = \{\simeq(rv_1)/V_1, \ldots, \simeq(rv_m)/V_m\}$. The latter replaces all evaluations of random variables on which the f_i depends by variables for integration.

$$\begin{aligned} P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n) &= \mathbb{E}[I^{x_1}_{rel_1}(t_{11}, t_{12}), \ldots, I^{x_n}_{rel_n}(t_{n1}, t_{n2})] \\ &= \int \cdots \int \left(I^{x_1}_{rel_1}(t_{11}\theta^{-1}, t_{12}\theta^{-1}) \cdots I^{x_n}_{rel_n}(t_{n1}\theta^{-1}, t_{n2})\theta^{-1} \right) d\mathscr{D}_{rv_1}(V_1) \cdots d\mathscr{D}_{rv_m}(V_m). \end{aligned} \tag{7}$$

Example 2 (*Basic distribution*)
Let $f_1, f_2, \ldots = dist\_lt(\simeq(b1), 3), dist\_lt(\simeq(b2), \simeq(b1)), \ldots$ Then the second distribution in the series is

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

$$P_F^{(2)}(dist\_lt(\simeq(b1),3)=x_1, dist\_lt(\simeq(b2),\simeq(b1))=x_2)$$
$$=\mathbb{E}[I^{x_1}_{\texttt{dist\_lt}}(\simeq(b1),3), I^{x_2}_{\texttt{dist\_lt}}(\simeq(b2),\simeq(b1))]$$
$$=\int\int \left(I^{x_1}_{\texttt{dist\_lt}}(V1,3), I^{x_2}_{\texttt{dist\_lt}}(V2,V1)\right) d\mathscr{D}_{b1}(V1) d\mathscr{D}_{b2}(V2).$$

By now we are able to prove the following proposition.

Proposition 1
Let P be a valid program. P defines a probability measure P_P over the set of fixpoints of operator T_P. Hence, P also defines for an arbitrary formula $\texttt{q}$ over atoms in its Herbrand base the probability that $\texttt{q}$ is true.

Proof sketch
It suffices to show that the series of distributions $P_F^{(n)}$ over facts (cf. (7)) is of the form that is required in the distribution semantics, that is, these are well-defined probability distributions that satisfy the compatibility condition, cf. (2). This is a direct consequence of the definition in terms of indicator functions and the measurability of the underlying facts required for valid programs. □

3.3 T_P semantics

In the following, we give a procedural view onto the semantics by extending T_P operator of Definition 1 to deal with probabilistic facts $\texttt{dist\_rel}(\texttt{t}_1,\texttt{t}_2)$. To do so, we introduce a function ReadTable(·) that keeps track of the sampled values of random variables to evaluate probabilistic facts. This is required because interpretations of a program only contain such probabilistic facts, but not the underlying outcomes of random variables. Given a probabilistic fact $\texttt{dist\_rel}(\texttt{t1},\texttt{t2})$, ReadTable returns the truth value of the fact based on the values of random variables h in the arguments, which are either retrieved from the table or sampled according to their definition $\texttt{h} \sim \mathscr{D}$ as included in the interpretation and stored in case they are not yet available.

Definition 4 (*Stochastic T_P operator*)
Let P be a valid program and $ground(P)$ be the set of all ground instances of clauses in P. Starting from a set of ground facts S, the ST_P operator returns

$$ST_P(S) := \Big\{ \texttt{h} \;\Big|\; \texttt{h :- b}_1,\ldots,\texttt{b}_\texttt{n} \in ground(P) \text{ and } \forall\ \texttt{b}_\texttt{i} :\ \text{either } \texttt{b}_\texttt{i} \in S \text{ or}$$
$$\left(\texttt{b}_\texttt{i} = dist\_rel(t1,t2) \wedge (t_j = \simeq(h) \rightarrow (\texttt{h} \sim \mathscr{D}) \in S) \wedge \right.$$
$$\left. \text{ReadTable}(b_i) = true\right)\Big\}.$$

ReadTable ensures that the basic facts are sampled from their joint distribution as defined in Section 3.2 during the construction of a standard fixpoint of logic program. Thus, each fixpoint of the ST_P operator corresponds to a possible world whose probability is given by the distribution semantics.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

4 Forward sampling using magic sets and backward reasoning

In this section we introduce our new method for probabilistic forward inference. To this aim, we first extend the magic set transformation to distributional clauses. Then we develop a rejection sampling scheme using this transformation. This scheme also incorporates backward reasoning to check for consistency with evidence during sampling and thus to reduce the rejection rate.

4.1 Probabilistic magic set transformation

The disadvantage of forward reasoning in logic programming is that the search is not goal-driven, which might generate irrelevant atoms. The *magic set* transformation (Bancilhon et al. 1986; Nilsson and Małiszyński 1996) focuses forward reasoning in logic programs toward a goal to avoid the generation of uninteresting facts. It thus combines the advantages of both reasoning directions.

Definition 5 (*Magic set transformation*)
If P is a logic program, then we use $\textsc{Magic}(P)$ to denote the smallest program such that if $\mathtt{A}_0 \text{ :- } \mathtt{A}_1, \ldots, \mathtt{A}_n \in P$, then

- $\mathtt{A}_0 \text{ :- } \mathtt{c}(\mathtt{A}_0), \mathtt{A}_1, \ldots, \mathtt{A}_n \in \textsc{Magic}(P)$ and
- for each $1 \leqslant i \leqslant n$: $\mathtt{c}(\mathtt{A}_i) \text{ :- } \mathtt{c}(\mathtt{A}_0), \mathtt{A}_1, \ldots, \mathtt{A}_{i-1} \in \textsc{Magic}(P)$.

The meaning of the additional $\mathtt{c}/1$ atoms (c=call) is that they "switch on" clauses when they are needed to prove a particular goal. If the corresponding switch for the head atom is not true, the body is not true and thus cannot be proven. The magic transformation is both sound and complete. Furthermore, if the SLD-tree of a goal is finite, forward reasoning in the transformed program terminates. The same holds if forward reasoning on the original program terminates.

We now extend this transformation to distributional clauses. The idea is that the distributional clause for a random variable h is activated when there is a call to a probabilistic fact $dist\_rel(t_1, t_2)$ depending on h.

Definition 6 (*Probabilistic magic set transformation*)
For program P, let P_L be P without distributional clauses. $\mathrm{M}(P)$ is the smallest program s.t. $\textsc{Magic}(P_L) \subseteq \mathrm{M}(P)$ and for each $\mathtt{h} \sim \mathscr{D} \text{ :- } \mathtt{b}_1, \ldots, \mathtt{b}_n \in P$ and $\mathtt{rel} \in \{\mathtt{eq}, \mathtt{lt}, \mathtt{leq}, \mathtt{gt}, \mathtt{geq}\}$:

- $\mathtt{h} \sim \mathscr{D} \text{ :- } (\mathtt{c}(\mathtt{dist\_rel}(\simeq(\mathtt{h}), \mathtt{X})); \mathtt{c}(\mathtt{dist\_rel}(\mathtt{X}, \simeq(\mathtt{h}))), \mathtt{b}_1, \ldots, \mathtt{b}_n. \in \mathrm{M}(P)$.
- $\mathtt{c}(\mathtt{b}_i) \text{ :- } (\mathtt{c}(\mathtt{dist\_rel}(\simeq(\mathtt{h}), \mathtt{X})); \mathtt{c}(\mathtt{dist\_rel}(\mathtt{X}, \simeq(\mathtt{h}))), \mathtt{b}_1, \ldots, \mathtt{b}_{i-1}. \in \mathrm{M}(P)$.

Then $\textsc{PMagic}(P)$ consists of the following:

- A clause $\mathtt{a\_p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \text{ :- } \mathtt{c}(\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)), \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ for each built-in predicate (including $\mathtt{dist\_rel}/2$ for $\mathtt{rel} \in \{\mathtt{eq}, \mathtt{lt}, \mathtt{leq}, \mathtt{gt}, \mathtt{geq}\}$) used in $\mathrm{M}(P)$.
- A clause $\mathtt{h} \text{ :- } \mathtt{b}'_1, \ldots, \mathtt{b}'_n$ for each clause $\mathtt{h} \text{ :- } \mathtt{b}_1, \ldots, \mathtt{b}_n \in \mathrm{M}(P)$, where $\mathtt{b}'_i = \mathtt{a\_b}_i$ if $\mathtt{b}_i$ uses a built-in predicate and else $\mathtt{b}'_i = \mathtt{b}_i$.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

Algorithm 1 Main loop for sampling-based inference to calculate the conditional probability $p(q|e)$ for query q, evidence e, and program L.

1: **function** EVALUATE(L, q, e, $Depth$)
2: $L^* :=$PMAGIC$(L) \cup \{c(a)|a \in e \cup q\}$
3: $n^+ := 0 \qquad n^- := 0$
4: **while** Not converged **do**
5: $(I, w) :=$STPMAGIC$(L^*, L, e, Depth)$
6: **if** $q \in I$ **then** $n^+ := n^+ + w$ **else** $n^- := n^- + w$
7: **return** $n^+/(n^+ + n^-)$

Note that every call to a built-in `b` is replaced by a call to `a_b`; the latter predicate is defined by a clause that is activated when there is a call to the built-in (`c(b)`) and that effectively calls the built-in. The transformed program computes the distributions only for random variables whose value is relevant to the query. These distributions are the same as those obtained in a forward computation of the original program. Hence, we can show the following.

Lemma 1
Let P be a program and PMAGIC(P) its probabilistic magic set transformation extended with a seed $c(q)$. The distribution over q defined by P and PMAGIC(P) is the same.

Proof sketch
In both programs, the distribution over q is determined by the distributions of the atoms $dist\_eq(t_1, t_2)$, $dist\_leq(t_1, t_2)$, $dist\_lt(t_1, t_2)$, $dist\_geq(t_1, t_2)$, and $dist\_gt(t_1, t_2)$ on which q depends in a forward computation of program P. The magic set transformation ensures that these atoms are called in the forward execution of PMAGIC(P). In PMAGIC(P), a call to such an atom activates the distributional clause for the involved random variable. As this distributional clause is a logic program clause, soundness and completeness of the magic set transformation ensures that the distribution obtained for that random variable is the same as in P. Hence, the distribution over q is same for both programs. □

4.2 Rejection sampling with heuristic lookahead

As discussed in Section 2.1, sampling-based approaches to probabilistic inference estimate the conditional probability $p(q|e)$ of a query q, given evidence e by randomly generating a large number of samples or possible worlds (cf. Algorithm 1). The algorithm starts by preparing the program L for sampling by applying the PMAGIC transformation. In the following we discuss our choice of subroutine STPMAGIC (cf. Algorithm 2), which realizes likelihood weighted sampling. It is used in Algorithm 1, line 5, to generate individual samples. It iterates the stochastic consequence operator of Definition 4 until either a fixpoint is reached or the current sample is inconsistent with the evidence. Finally, if the sample is inconsistent with the evidence, it receives weight 0.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

Algorithm 2 Sampling one interpretation; used in Algorithm 1.

```
1:  function STPMAGIC(L*, L, e, Depth)
2:      T_pf := ∅, T_dis := ∅, w := 1, I_old := ∅, I_new := ∅
3:      repeat
4:          I_old := I_new
5:          for all (h :- body) ∈ L* do
6:              split body in B_PF (prob. facts) and B_L (the rest)
7:              for all grounding substitution θ such that B_Lθ ⊆ I_old do
8:                  s := true, w_d := 1
9:                  while s ∧ B_PF ≠ ∅ do
10:                     select and remove pf from B_PF
11:                     (b_pf, w_pf) := READTABLE(pfθ, I_old, T_pf, T_dis, L, e, Depth)
12:                     s := s ∧ b_pf          w_d := w_d · w_pf
13:                 if s then
14:                     if hθ ∈ e⁻ then return (I_new, 0)        ▷ check negative evidence
15:                     I_new := I_new ∪ {hθ}          w := w · w_d
16:     until I_new = I_old ∨ w = 0                  ▷ Fixpoint or impossible evidence
17:     if e⁺ ⊆ I_new then return (I_new, w)                ▷ check positive evidence
18:     else return (I_new, 0)
```

Algorithm 3 details the procedure used in line 11 of Algorithm 2 to sample from a given distributional clause. The function READTABLE returns the truth value of the probabilistic fact, together with its weight. If the outcome is not yet tabled, it is computed. Note that `false` is returned when the outcome is not consistent with the evidence. Involved distributions, if not yet tabled, are sampled in line 5. In the infinite case, SAMPLE simply returns the sampled value. In the finite case, it is directed toward generating samples that are consistent with the evidence. Firstly, all possible choices that are inconsistent with the negative evidence are removed. Secondly, when there is positive evidence for a particular value, only that value is left in the distribution. Thirdly, it is checked whether each left value is consistent with all other evidence. This consistency check is performed by a simple depth-bounded meta-interpreter. For positive evidence, it attempts a top-down proof of the evidence atom in the original program using the function MAYBEPROOF. Subgoals for which the depth-bound is reached, as well as probabilistic facts that are not yet tabled are assumed to succeed. If this results in a proof, the value is consistent, otherwise it is removed. Similarly, for negative evidence: in MAYBEFAIL, subgoals for which the depth-bound is reached, as well as probabilistic facts that are not yet tabled are assumed to fail. If this results in failure, the value is consistent, otherwise it is removed. The *Depth* parameter allows one to trade the computational cost associated with this consistency check for a reduced rejection rate.

Note that the modified distribution is normalized and the weight is adjusted in each of these three cases. The weight adjustment takes into account the removed elements that cannot be sampled and it is necessary because it can depend on the distributions sampled so far and the elements removed from the distribution

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

Algorithm 3 Evaluating a probabilistic fact pf used in Algorithm 2. ComputePF(pf, T_{dis}) computes the truth value and the probability of pf according to the information in T_{dis}.

function ReadTable($pf, I, T_{pf}, T_{dis}, L, e, Depth$)
  if $pf \notin T_{pf}$ **then**
    for all random variable h occurring in pf where $h \notin T_{dis}$ **do**
      if $h \sim D \notin I$ **then return** $(false, 0)$
      if not Sample($h, D, T_{dis}, I, L, e, Depth$) **then return** $(false, 0)$
    $(b, w) :=$ ComputePF(pf, T_{dis})
    if $(b \wedge (pf \in e^-)) \vee (\neg b \wedge (pf \in e^+))$ **then**
      return $(false, 0)$ ▷ inconsistent with evidence
    extend T_{pf} with (pf, b, w)
  return (b, w) as stored in T_{pf} for pf
procedure Sample(h, $\mathscr{D}$, T_{dis}, I, L, e, $Depth$)
  $w_h := 1$, $\mathscr{D}' := \mathscr{D}$ ▷ Initial weight, temp. distribution
  if $\mathscr{D}' = [p_1 : \mathtt{a}_1, \ldots, p_n : \mathtt{a}_n]$ **then** ▷ finite distribution
    for $p_j : \mathtt{a}_j \in \mathscr{D}'$ where dist_eq(h, $\mathtt{a}_j$) $\in e^-$ **do** ▷ remove neg. evidence
      $\mathscr{D}' :=$ Norm($\mathscr{D}' \setminus \{p_j : \mathtt{a}_j\}$), $w_h := w_h \times (1 - p_j)$
    if $\exists v :$ dist_eq($\simeq$(h), v) $\in e^+$ and $p : \mathtt{v} \in \mathscr{D}'$ **then**
      $\mathscr{D}' := [1 : v]$, $w_h := w_h \times p$
    for $p_j : \mathtt{a}_j \in \mathscr{D}'$ **do** ▷ remove choices that make e^+ impossible
      if $\exists b \in e^+$: not MaybeProof($b, Depth, I \cup \{\mathtt{dist\_eq(h, a_j)}\}, L$) **or**
        $\exists b \in e^-$: not MaybeFail($b, Depth, I \cup \{\mathtt{dist\_eq(h, a_j)}\}, L$) **then**
          $\mathscr{D}' :=$ Norm($\mathscr{D}' \setminus \{p_j : \mathtt{a}_j\}$), $w_h := w_h \times (1 - p_j)$
  if $\mathscr{D}' = \emptyset$ **return** false
  Sample x according to $\mathscr{D}'$, extend T_{dis} with (h, x) and **return** true

sampled in Sample (the clause bodies of the distribution clause are instantiating the distribution).

5 Experiments

We implemented our algorithm in YAP Prolog and set up experiments to answer the following questions:

Q 1. Does the look ahead-based sampling improve the performance?
Q 2. How do rejection sampling and likelihood weighting compare?

To answer the first question, we used the distributional program in Figure 1, which models an urn containing a random number of balls. The number of balls is uniformly distributed from 1 to 10 and each ball is either red or green with equal probability. We draw a ball with replacement from the urn for 8 times and observe its color. We also define the atom nogreen(D) to be true if and only if we did not draw any green ball in draw 1 to D. We evaluated the query

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

```
numballs ~ uniform([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).
ball(M) :- between(1, numballs, M).
color(B) ~ uniform([red, green]) :- ball(B).
draw(N) :- between(1, 8, N).
nogreen(0).
nogreen(D) :- dist_eq(≃ (color(≃ (drawnball(D)))), red), D2 is D − 1, nogreen(D2).
drawnball(D) ~ uniform(L) :- draw(D),findall(B, ball(B), L).
```

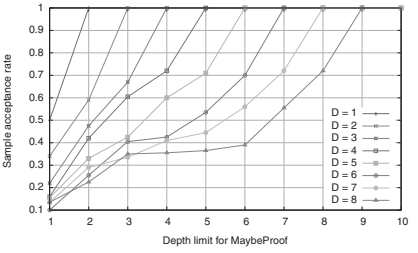

Fig. 1. The program modeling the urn (left); rate of accepted samples (right) for evaluating the query $P(\texttt{dist\_eq}(\simeq(\texttt{color}(\simeq(\texttt{drawnball}(1)))), \texttt{red}) \mid \texttt{nogreen}(D))$ for $D = 1, 2, \ldots, 10$ and for $Depth = 1, 2, \ldots, 8$ using Algorithm 1. The acceptance rate is calculated by generating 200 samples using our algorithm and counting the number of samples with weight larger than 0.

$P(\texttt{dist\_eq}(\simeq(\texttt{color}(\simeq(\texttt{drawnball}(1)))), \texttt{red}) \mid \texttt{nogreen}(D))$ for $D = 1, 2, \ldots, 8$. Note that the evidence implies that the first drawn ball is red, hence the probability of the query is 1; however, the number of steps required to prove that the evidence is inconsistent with drawing a green first ball increases with D, so as D becomes larger, larger *Depth* is required to reach a 100% acceptance rate for the sample as illustrated in Figure 1. It is clear that by increasing the *Depth* limit, each sample will take longer to be generated. Thus, the *Depth* parameter allows one to trade off convergence speed of the sampling and the time each sample needs to be generated. Depending on the program, the query, and the evidence there is an optimal depth for the lookahead.

To answer the second question, we used the standard example for BLOG (Milch *et al.* 2005). An urn contains an unknown number of balls where every ball can be either green or blue with $p = 0.5$. When drawing a ball from the urn, we observe its color but do not know which ball it is. When we observe the color of a particular ball, there are 20% chances to observe the wrong color, e.g., green instead of blue. We have some prior belief over the number of balls in the urn. If 10 balls are drawn with replacement from the urn and we saw 10 times the color green, what is the probability that there are n balls in the urn? We consider two different prior distributions: in the first case, the number of balls is uniformly distributed between 1 and 8, in the second case, it is Poisson-distributed with mean $\lambda = 6$.

One run of the experiment corresponds to sampling the number N of balls in the urn, the color for each of the N balls, and for each of the 10 draws for which the ball is drawn and whether or not the color is observed correctly in this draw. Once these values are fixed, the sequence of observed colors is determined. This implies that for a fixed number N of balls, there are $2^N \times N^{10}$ possible proofs. In case of uniform distribution, exact PRISM inference can be used to calculate the probability for the given number of balls, with a total runtime of 0.16 seconds for all eight cases. In the case of the Poisson distribution, this is only possible up to 13 balls,

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

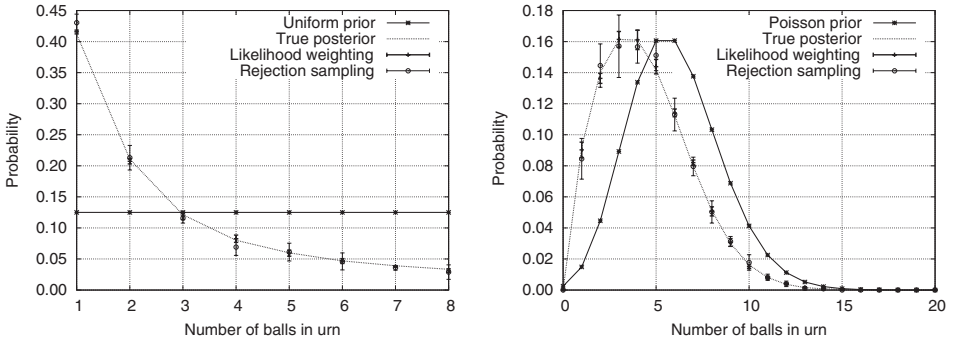

Fig. 2. Results of the urn experiment with forward reasoning. Ten balls with replacement were drawn and each time green was observed. Left: Uniform prior over number of balls; right: Poisson prior ($\lambda = 6$).

as with more balls PRISM runs out of memory. For inference using sampling, we generate 20,000 samples with the uniform prior, and 100,000 with the Poisson prior. We report average results over five repetitions. For these priors, PRISM generates 8,015 and 7,507 samples per second, ProbLog backward sampling generates 708 and 510 samples, BLOG generates 3,008 and 2,900 samples, and our new forward sampling (with rejection sampling) generates 760 and 731 samples per second. The results using our algorithm for both rejection sampling and likelihood weighting with $Depth = 0$ are shown in Figure 2. As the graphs show, the standard deviation for rejection sampling is much larger than for likelihood weighting.

6 Conclusions and related work

We have contributed a novel construct for probabilistic logic programming, the distributional clauses, and defined its semantics. Distributional clauses allow one to represent continuous variables and to reason about an unknown number of objects. In this regard, this construct is similar in spirit to languages such as BLOG and Church, but it is strongly embedded in a logic programming context. This embedding allowed us to propose a novel inference method based on a combination of importance sampling and forward reasoning. This contrasts with the majority of probabilistic logic programming languages, which are based on backward reasoning (possibly enhanced with tabling (Sato and Kameya 2001; Mantadelis and Janssens 2010)). Furthermore, only few of these techniques employ sampling, but see Kimmig *et al.* (2011) for the Monte Carlo approach using backward reasoning. Another key difference with the existing probabilistic logic programming approaches is that the described inference method can handle evidence. This is due to the magic set transformation that targets the generative process toward the query and evidence and instantiates only relevant random variables.

P-log (Baral et al. 2009) is a probabilistic language based on Answer Set Prolog (ASP). It uses a standard ASP solver for inference and is thus based on forward reasoning, but without use of sampling. Magic sets are also used in probabilistic Datalog (Fuhr 2000) as well as in Dyna, a probabilistic logic programming language

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press

(Eisner *et al.* 2005) based on rewrite rules that uses forward reasoning. However, neither of them uses sampling. Furthermore, Dyna and PRISM require the exclusive-explanation assumption. This assumption states that no two different proofs for the same goal can be true simultaneously, that is, they have to rely on at least one basic random variable with different outcomes. Distributional clauses (and the ProbLog language) do not impose such a restriction. Other related work includes MCMC-based sampling algorithms, such as the approach for SLP (Angelopoulos and Cussens 2003). Church's inference algorithm is also based on MCMC, and BLOG is also able to employ MCMC. At least for BLOG it seems to be required to define the domain-specific proposal distribution for fast convergence. With regard to future work, it would be interesting to consider evidence on continuous distributions, as it is currently restricted to finite distribution. Program analysis and transformation techniques to further optimize the program with respect to the evidence and query could be used to increase the sampling speed. Finally, the implementation could be optimized by memoizing some information from previous runs and use it to more rapidly prune as well as sample.

Acknowledgements

Angelika Kimmig and Bernd Gutmann are supported by the Research Foundation-Flanders (FWO-Vlaanderen). This work is supported by the GOA project 2008/08 Probabilistic Logic Learning and the European Community's Seventh Framework Programme under grant agreement First-MM-248258.

References

Angelopoulos, N. and Cussens, J. 2003. Prolog issues and experimental results of an MCMC algorithm. In *Web Knowledge Management and Decision Support*, O. Bartenstein, U. Geske, M. Hannebauer and O. Yoshie, Eds. Lecture Notes in Computer Science, vol. 2543. Springer, Berlin, Germany 186–196.

Bancilhon, F., Maier, D., Sagiv, Y. and Jeffrey D. Ullman. 1986. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986)*. ACM, Cambridge, MA, 1–15.

Baral, C., Gelfond, M. and Rushton, N. 2009. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming 9*, 1, 57–144.

De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I. and Vennekens, J. 2008. Towards digesting the alphabet-soup of statistical relational learning. In *Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications*, D. Roy, J. Winn, D. McAllester, V. Mansinghka and J. Tenenbaum, Eds. Whistler, British Columbia, Canada.

De Raedt, L., Frasconi, P., Kersting, K. and Muggleton, S. 2008. *Probabilistic Inductive Logic Programming – Theory and Applications*. LNCS, vol. 4911. Springer, Berlin, Germany.

De Raedt, L., Kimmig, A. and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (*IJCAI*) Hyderabad, India, January 6–12. 2462–2467.

Eisner, J., Goldlust, E. and Smith, N. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing* (*HLT/EMNLP-05*), Vancouver, BC, Canada, October 6–8.

Fuhr, N. 2000. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science (JASIS) 51*, 2, 95–110.

Getoor, L. and Taskar, B. 2007. *An Introduction to Statistical Relational Learning*. MIT Press, Cambridge, MA.

Gogate, V. and Dechter, R. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence 175*, 694–729.

Goodman, N., Mansinghka, V. K., Roy, D. M., Bonawitz, K. and Tenenbaum, J. B. 2008. Church: A language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence,* Helsinki, Finland, July 9–12. AUAI Press, Helsinki, Finland, 220–229.

Gutmann, B., Jaeger, M. and De Raedt, L. 2010. Extending ProbLog with continuous distributions. In *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)*, Firenze, Italy, June 27–30, P. Frasconi and F. A. Lisi, Eds. Springer, Berlin, Germany.

Kersting, K. and De Raedt, L. 2008. Basic principles of learning Bayesian logic programs. See De Raedt *et al*. (2008), 189–221.

Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V. and Rocha, R. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming (TPLP) 11*, 235–262.

Koller, D. and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA.

Mantadelis, T. and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP-10)*, M. V. Hermenegildo and T. Schaub, Eds. LIPIcs, vol. 7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 124–133.

Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. and Kolobov, A. 2005. BLOG: Probabilistic models with unknown objects. In *Proceedings of THE Nineteenth International Joint Conference on Artificial Intelligence* (*IJCAI*), Edinburgh, Scotland, UK, July 30–August 5. Springer Verlag, Berlin, Germany, 1352–1359.

Milch, B., Marthi, B., Sontag, D., Russell, S., Ong, D. L. and Kolobov, A. 2005. Approximate inference for infinite contingent Bayesian networks. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, January 6–8, Barbados*, R. G. Cowell and Z. Ghahramani, Eds. Society for Artificial Intelligence and Statistics, New Jersey, 238–245. (Available electronically at http://www.gatsby.ucl.ac.uk/aistats/)

Nilsson, U. and Małiszyński, J. 1996. *Logic, Programming And Prolog*, 2nd ed. Wiley, Hoboken, NJ.

Pfeffer, A. 2001. IBAL: A probabilistic rational programming language. In *Seventeenth International Joint Conference on Artificial Intelligence* (IJCAI), Seattle, Washington, August 4–10. Springer, Berlin, Germany, 733–740.

Poole, D. 2008. The independent choice logic and beyond. In *Probabilistic Inductive Logic Programming – Theory and Applications*, L. De Raedt, P. Frasconi, K. Kersting and S. Muggleton, Eds. LNCS, vol. 4911. Springer, Berlin, Germany, 222–243.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning 62*, 1–2, 107–136.

Santos Costa, V., Page, D. and Cussens, J. 2008. CLP(*BN*): Constraint logic programming for probabilistic knowledge. See De Raedt *et al*. (2008), 156–188.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the Twelth International Conference on Logic Programming (ICLP 1995)*. MIT Press, Cambridge, MA, 715–729.

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Resesearch (JAIR) 15*, 391–454.

Vennekens, J., Denecker, M. and Bruynooghe, M. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming 9*, 3, 245–308.

Wasserman, L. 2003. *All of Statistics: A Concise Course in Statistical Inference (Springer Texts in Statistics)*. Springer, Berlin, Germany.

https://doi.org/10.1017/S1471068411000238 Published online by Cambridge University Press