Probabilistic Programming Concepts

Luc De Raedt Angelika Kimmig
Department of Computer Science, KU Leuven
Celestijnenlaan 200a – box 2402, 3001 Heverlee, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract

A multitude of different probabilistic programming languages exists today, all extending a traditional programming language with primitives to support modeling of complex, structured probability distributions. Each of these languages employs its own probabilistic primitives, and comes with a particular syntax, semantics and inference procedure. This makes it hard to understand the underlying programming concepts and appreciate the differences between the different languages.

To obtain a better understanding of probabilistic programming, we identify a number of core programming concepts underlying the primitives used by various probabilistic languages, discuss the execution mechanisms that they require and use these to position state-of-the-art probabilistic languages and their implementation.

While doing so, we focus on probabilistic extensions of *logic* programming languages such as Prolog, which have been developed since more than 20 years.

1 Introduction

The vast interest in statistical relational learning [Getoor and Taskar, 2007], probabilistic (inductive) logic programming [De Raedt et al., 2008] and probabilistic programming languages [Roy et al., 2008] has resulted in a wide variety of different formalisms, models and languages. The multitude of probabilistic languages that exists today provides evidence for the richness and maturity of the field, but on the other hand, makes it hard to get an appreciation and understanding of the relationships and differences between the different languages. Furthermore, most arguments in the literature about the relationship amongst these languages are about the expressiveness of these languages, that is, they state (often in an informal way) that one language is more expressive than another one (implying that the former could be used to emulate the latter). By now, it is commonly accepted that the more interesting question is concerned with the underlying concepts that these languages employ and their effect on the inference mechanisms, as their expressive power is often very similar. However, a multitude of different probabilistic primitives exists, which makes it hard to appreciate their relationships.[1]

[1]Throughout the paper we use the term *primitive* to denote a particular syntactic and semantic construct that is available in a particular probabilistic programming language, and

To alleviate these difficulties and obtain a better understanding of the field we identify a number of core probabilistic programming concepts and relate them to one another. We cover the basic concepts representing different types of random variables, but also general modeling concepts such as negation or time and dynamics, and programming constructs such as meta-calls and ways to handle sets. While doing so, we focus on probabilistic extensions of logic programming languages because this is (arguably) the first and best studied probabilistic programming paradigm. It has been studied for over 20 years starting with the seminal work of David Poole [1992] and Taisuke Sato [1995], and now includes languages such as CLP($\mathcal{BN}$) [Santos Costa et al., 2008], BLPs [Kersting and De Raedt, 2008], ICL [Poole, 2008], PRISM [Sato and Kameya, 2001], ProbLog [De Raedt et al., 2007], LPADs [Vennekens et al., 2004], CP-logic [Vennekens et al., 2009], SLPs [Muggleton, 1996], PROPPR [Wang et al., 2013], P-log [Baral et al., 2009] and Dyna [Eisner et al., 2005]. Another reason for focussing on probabilistic extensions of logic programming languages is that the concepts are all embedded within the same host language, so we can focus on semantics rather than syntax. At the same time, we also relate the concepts to alternative probabilistic programming languages such as Church [Goodman et al., 2008], IBAL [Pfeffer, 2001], Figaro [Pfeffer, 2009] and BLOG [Milch et al., 2005] and to some extent also to statistical relational learning models such as RBNs [Jaeger, 2008], Markov logic [Richardson and Domingos, 2006], and PRMs [Getoor et al., 2007]. Most statistical relational learning approaches employ a knowledge-based model construction approach, in which the logic is used as a template for constructing a graphical model. Typical probabilistic programming languages, on the other hand, employ a variant of Sato's distribution semantics [Sato, 1995], in which random variables directly correspond to ground facts and a traditional program specifies how to deduce further knowledge from these facts. This difference explains why we introduce the concepts in the context of the distribution semantics, and discuss approaches to knowledge-based model construction separately.

Inference is a key challenge in probabilistic programming and statistical relational learning. Furthermore, the choice of inference approach often influences which probabilistic primitives can be supported. Enormous progress has been made in the past few years w.r.t. probabilistic inference and numerous inference procedures have been contributed. Therefore, we also identify some core classes of inference mechanisms for probabilistic programming and discuss which ones to use for which probabilistic concept. Inference in probabilistic languages also is an important building block of approaches that learn the structure and/or parameters of such models from data. Given the variety of approaches that exist today, a discussion of learning is beyond the scope of this paper.

To summarize, the key contributions of this paper are (1) the identification of a number of core concepts that are used by various probabilistic languages, (2) a discussion of the execution mechanisms that they require, and (3) a positioning of state-of-the-art probabilistic languages and implementations w.r.t. these concepts. Although many of the concepts we discuss are well-described in the literature, some even in survey papers [De Raedt and Kersting, 2003, Poole, 2008], we believe a new and up-to-date survey is warranted due to the rapid

the term *concept* to denote the underlying notion. Different primitives may hence realize the same concept.

developments of the field which rapidly renders existing surveys incomplete and even outdated. To the best of our knowledge, this is also the first time that such a wide variety of probabilistic programming concepts and languages, also in connection to inference, is discussed in a single paper.

We expect the reader to be familiar with basic language concepts and terms of Prolog [Lloyd, 1989, Flach, 1994]; a quick summary can be found in Appendix A.

This paper is organized as follows. We first discuss the distribution semantics (Section 2) and classify corresponding inference approaches according to their logical and probabilistic components (Section 3). Section 4 identifies the probabilistic programming concepts. In Section 5, we discuss the relation with statistical relational modeling approaches rooted in graphical models. Section 6 relates the different inference approaches to the probabilistic programming concepts.

2 Distribution Semantics

Sato's distribution semantics [Sato, 1995] is a well-known semantics for probabilistic logics that has been used many times in the literature, cf. [Dantsin, 1991, Poole, 1993, Fuhr, 2000, Poole, 2000, Sato and Kameya, 2001, Dalvi and Suciu, 2004, De Raedt et al., 2007]. Prominent examples of Prolog-based languages using this semantics include ICL [Poole, 2008], PRISM [Sato and Kameya, 2001] and ProbLog [De Raedt et al., 2007, Kimmig et al., 2011a], even though there exist subtle differences between these languages as we will illustrate later. Sato has defined the distribution semantics for a countably infinite set of random variables and a general class of distributions. We focus on the finite case here, discussing the two most popular instances of the semantics, based on a set of independent random variables and independent probabilistic choices, respectively, and refer to [Sato, 1995] for details on the general case.

2.1 Probabilistic Facts

The arguably most basic instance of the distribution semantics uses a finite set of Boolean random variables that are all pairwise independent. Throughout the paper, we use the following running example inspired by the well-known alarm Bayesian network:

$$
\begin{aligned}
0.1 :: \texttt{burglary}. \quad & 0.7 :: \texttt{hears\_alarm}(\texttt{mary}). \\
0.2 :: \texttt{earthquake}. \quad & 0.4 :: \texttt{hears\_alarm}(\texttt{john}). \\
\texttt{alarm} &:\!- \texttt{earthquake}. \\
\texttt{alarm} &:\!- \texttt{burglary}. \\
\texttt{calls}(\texttt{X}) &:\!- \texttt{alarm}, \texttt{hears\_alarm}(\texttt{X}). \\
\texttt{call} &:\!- \texttt{calls}(\texttt{X}).
\end{aligned}
\tag{1}
$$

The program consists of a set R of definite clauses or *rules*, and a set F of ground facts f, each of them labeled with a probability p, written as $p :: f$. We call such labeled facts *probabilistic facts*. Each probabilistic fact corresponds to

a Boolean random variable that is *true* with probability p and *false* with probability $1-p$. We use b, e, hm and hj to denote the random variables corresponding to `burglary`, `earthquake`, `hears_alarm(mary)` and `hears_alarm(john)`, respectively. Assuming that all these random variables are independent, we obtain the following probability distribution P_F over truth value assignments to these random variables and their corresponding sets of ground facts $F' \subseteq F$:

$$P_F(F') = \prod_{f_i \in F'} p_i \cdot \prod_{f_i \in F \setminus F'} (1 - p_i) \tag{2}$$

For instance, the truth value assignment `burglary` = true, `earthquake` = false, `hears_alarm(mary)` = true, `hears_alarm(john)` = false, which we will abbreviate as $b \wedge \neg e \wedge hm \wedge \neg hj$, corresponds to the set of facts {`burglary`, `hears_alarm(mary)`}, and has probability $0.1 \cdot (1-0.2) \cdot 0.7 \cdot (1-0.6) = 0.0336$. The corresponding logic program obtained by adding the set of rules R to the set of facts, also called a *possible world*, is

```
                burglary.
hears_alarm(mary).
           alarm :- earthquake.
           alarm :- burglary.
       calls(X) :- alarm, hears_alarm(X).
            call :- calls(X).
```
(3)

As each logic program obtained by fixing the truth values of all probabilistic facts has a unique least Herbrand model, P_F can be used to define the *success probability* of a query q, that is, the probability that q is true in a randomly chosen such program, as the sum over all programs that entail q:

$$P_s(q) := \sum_{\substack{F' \subseteq F \\ \exists \theta F' \cup R \models q\theta}} P_F(F') \tag{4}$$

$$= \sum_{\substack{F' \subseteq F \\ \exists \theta F' \cup R \models q\theta}} \prod_{f_i \in F'} p_i \cdot \prod_{f_i \in F \setminus F'} (1 - p_i)\ . \tag{5}$$

Naively, the success probability can thus be computed by enumerating all sets $F' \subseteq F$, for each of them checking whether the corresponding possible world entails the query, and summing the probabilities of those that do. As fixing the set of facts yields an ordinary logic program, the entailment check can use any reasoning technique for such programs.

For instance, *forward reasoning*, also known as applying the T_P operator, starts from the set of facts and repeatedly uses rules to derive additional facts until no more facts can be derived. In our example possible world (3), we thus start from {`burglary`, `hears_alarm(mary)`}, and first add `alarm` due to the second rule based on `burglary`. This in turn makes it possible to add `calls(mary)` using the third rule and substitution `X=mary`, and finally, `call` is added using the last rule, resulting in the least Herbrand model {`burglary`, `hears_alarm(mary)`, `alarm`, `calls(mary)`, `call`}. This possible world thus contributes to the success probabilities of `alarm`, `calls(mary)` and `call`, but not to the one of `calls(john)`.

world	`calls(john)`	probability
$b \wedge \neg e \wedge hm \wedge \neg hj$	false	$0.1 \cdot (1-0.2) \cdot 0.7 \cdot (1-0.4) = 0.0336$
$b \wedge \neg e \wedge hm \wedge hj$	true	$0.1 \cdot (1-0.2) \cdot 0.7 \cdot 0.4 = 0.0224$
$b \wedge e \wedge hm \wedge \neg hj$	false	$0.1 \cdot 0.2 \cdot 0.7 \cdot (1-0.4) = 0.0084$
$b \wedge e \wedge hm \wedge hj$	true	$0.1 \cdot 0.2 \cdot 0.7 \cdot 0.4 = 0.0056$
$\neg b \wedge e \wedge hm \wedge \neg hj$	false	$(1-0.1) \cdot 0.2 \cdot 0.7 \cdot (1-0.4) = 0.0756$
$\neg b \wedge e \wedge hm \wedge hj$	true	$(1-0.1) \cdot 0.2 \cdot 0.7 \cdot 0.4 = 0.0504$

Table 1: The possible worlds of program (1) where `calls(mary)` is true.

An alternative to forward reasoning is *backward reasoning*, also known as SLD-resolution or proving, which we again illustrate for our example possible world (3). It starts from a given query, e.g., `call`, and uses the rules in the opposite direction: in order to prove a fact appearing in the head of a clause, we have to prove all literals in the clause's body. For instance, based on the last rule, to prove `call`, we need to prove `calls(X)` for some instantiation of `X`. Using the third rule, this means proving `alarm, hears_alarm(X)`. To prove `alarm`, we could use the first rule and prove `earthquake`, but this fails for our choice of facts, as there is no rule (or fact) for the latter. We thus *backtrack* to the second rule for `alarm`, which requires proving `burglary`, which is proven by the corresponding fact. Finally, we prove `hears_alarm(X)` using the fact `hears_alarm(mary)`, substituting `mary` for `X`, which completes the proof for `call`.

Going over all possible worlds in this way, we obtain the success probability of `calls(mary)`, $P_s(\texttt{calls(mary)}) = 0.196$, as the sum of the probabilities of six possible worlds (listed in Table 1).

Clearly, enumerating all possible worlds is infeasible for larger programs; we will discuss alternative inference techniques from the literature in Section 3.

For ease of modeling (and to allow for countably infinite sets of probabilistic facts), probabilistic languages such as ICL and ProbLog use *non-ground probabilistic facts* to define sets of random variables. All ground instances of such a fact are mutually independent and share the same probability value. As an example, consider a simple coin game which can be won either by throwing two times heads or by cheating. This game can be modeled by the program below. The probability to win the game is then defined by the success probability $P_s(\texttt{win})$.

```
0.5 :: heads(X).                    0.2 :: cheat_successfully.
win :- cheat_successfully.
win :- heads(1), heads(2).
```

Legal groundings of such facts can also be restricted by providing a domain, as in the following variant of our alarm example where all persons have the same

probability of independently hearing the alarm:

$$
\begin{aligned}
0.1 :: \texttt{burglary}. \quad 0.2 :: \texttt{earthquake}\\
0.7 :: \texttt{hears\_alarm(X)} :- \texttt{person(X)}.\\
\texttt{person(mary)}. \quad \texttt{person(john)}. \quad \texttt{person(bob)}. \quad \texttt{person(ann)}.\\
\texttt{alarm} :- \texttt{earthquake}.\\
\texttt{alarm} :- \texttt{burglary}.\\
\texttt{calls(X)} :- \texttt{alarm}, \texttt{hears\_alarm(X)}.\\
\texttt{call} :- \texttt{calls(X)}.
\end{aligned}
$$

If such domains are defined purely logically, without using probabilistic facts, the basic distribution is still well defined.

It is often assumed that probabilistic facts do not unify with other probabilistic facts or heads of rules.

2.2 Probabilistic Choices

As already noted by Sato [1995], probabilistic facts (or binary switches) are expressive enough to represent a wide range of models, including Bayesian networks, Markov chains and hidden Markov models. However, for ease of modeling, it is often more convenient to use multi-valued random variables instead of binary ones. The concept commonly used to realize such variables in the distribution semantics is a probabilistic choice, that is, a finite set of ground atoms exactly one of which is true in any possible world. Examples of such choices are the *probabilistic alternatives* of the Independent Choice Logic (ICL) [Poole, 2000] and probabilistic Horn abduction (PHA) [Poole, 1993], the *multi-ary random switches* of PRISM [Sato and Kameya, 2001], the *probabilistic clauses* of stochastic logic programs (SLPs) [Muggleton, 1996], and the *annotated disjunctions* of logic programs with annotated disjunctions (LPADs) [Vennekens et al., 2004], or the *CP-events* of CP-logic [Vennekens, 2007]. These are all closely related, e.g., the probabilistic clauses of SLPs map onto the switches of PRISM [Cussens, 2005], and the probabilistic alternatives of ICL onto annotated disjunctions (and vice versa) [Vennekens et al., 2004]. We therefore restrict the following discussion to annotated disjunctions [Vennekens et al., 2004], using the notation introduced below.

An *annotated disjunction* (AD) is an expression of the form

$$\mathtt{p}_1 :: \mathtt{h}_1;\ \ldots\ ; \mathtt{p}_N :: \mathtt{h}_N :- \mathtt{b}_1,\ \ldots\ , \mathtt{b}_M.$$

where $\mathtt{b}_1, \ldots, \mathtt{b}_M$ is a possibly empty conjunction of literals, the p_i are probabilities and $\sum_{i=1}^{N} p_i \leq 1$. Considered in isolation, an annotated disjunction states that if the body $\mathtt{b}_1, \ldots, \mathtt{b}_M$ is true at most one of the $\mathtt{h}_i$ is true as well, where the choice is governed by the probabilities (see below for interactions between multiple ADs with unifying atoms in the head). If the p_i in an annotated disjunction do not sum to 1, there is also the case that nothing is chosen. The probability of this event is $1 - \sum_{i=1}^{n} p_i$. A probabilistic fact is thus a special case of an AD with a single head atom and empty body.

For instance, consider the following program:

$$0.4 :: \texttt{draw}.$$
$$\frac{1}{3} :: \texttt{color}(\texttt{green}); \frac{1}{3} :: \texttt{color}(\texttt{red}); \frac{1}{3} :: \texttt{color}(\texttt{blue}) :- \texttt{draw}.$$

The probabilistic fact states that we draw a ball from an urn with probability 0.4, and the annotated disjunction states that if we draw a ball, the color is picked uniformly among `green`, `red` and `blue`. As for probabilistic facts, a non-ground AD denotes the set of all its groundings, and for each such grounding, choosing one of its head atoms to be true is seen as an independent random event. That is, the annotated disjunction

$$\frac{1}{3} :: \texttt{color}(\texttt{B}, \texttt{green}); \frac{1}{3} :: \texttt{color}(\texttt{B}, \texttt{red}); \frac{1}{3} :: \texttt{color}(\texttt{B}, \texttt{blue}) :- \texttt{ball}(\texttt{B}).$$

defines an independent probabilistic choice of color for each ball `B`.

As noted already by Vennekens et al. [2004], the probabilistic choice over head atoms in an annotated disjunction can equivalently be expressed using a set of logical clauses, one for each head, and a probabilistic choice over facts added to the bodies of these clauses, e.g.

$$\begin{aligned}
\texttt{color}(\texttt{B}, \texttt{green}) &:- \texttt{ball}(\texttt{B}), \texttt{choice}(\texttt{B}, \texttt{green}).\\
\texttt{color}(\texttt{B}, \texttt{red}) &:- \texttt{ball}(\texttt{B}), \texttt{choice}(\texttt{B}, \texttt{red}).\\
\texttt{color}(\texttt{B}, \texttt{blue}) &:- \texttt{ball}(\texttt{B}), \texttt{choice}(\texttt{B}, \texttt{blue}).\\
\frac{1}{3} :: \texttt{choice}(\texttt{B}, \texttt{green}); \frac{1}{3} &:: \texttt{choice}(\texttt{B}, \texttt{red}); \frac{1}{3} :: \texttt{choice}(\texttt{B}, \texttt{blue}).
\end{aligned}$$

This example illustrates that annotated disjunctions define a distribution P_F over basic facts as required in the distribution semantics, but can simplify modeling by directly expressing probabilistic consequences.

As mentioned above, a probabilistic fact directly corresponds to an annotated disjunction with a single atom in the head and an empty body. Conversely, each annotated disjunction can – for the purpose of calculating success probabilities – be equivalently represented using a set of probabilistic facts and deterministic clauses, which together simulate a sequential choice mechanism; we refer to Appendix B for details.

Independent Causes Some languages, e.g. ICL [Poole, 2008], assume that head atoms in the same or different annotated disjunctions cannot unify with one another, while others, e.g., LPADs [Vennekens et al., 2004], do not make this restriction, but instead view each annotated disjunction as an independent cause for the conclusions to hold. In that case, the structure of the program defines the combined effect of these causes, similarly to how the two clauses for `alarm` in our earlier example (1) combine the two causes `burglary` and `earthquake`. We illustrate this on the Russian roulette example by Vennekens et al. [2009], which involves two guns.

$$\begin{aligned}
\tfrac{1}{6} :: \texttt{death} :- \ & \texttt{pull_trigger}(\texttt{left_gun}).\\
\tfrac{1}{6} :: \texttt{death} :- \ & \texttt{pull_trigger}(\texttt{right_gun}).
\end{aligned}$$

Each gun is an independent cause for death. Pulling both triggers will result in death being true with a probability of $1 - (1 - \frac{1}{6})^2$, which exactly corresponds

to the probability of `death` being proven via the first or via the second annotated disjunction (or both). Assuming independent causes closely corresponds to the noisy-or combining rule that is often employed in the Bayesian network literature, cf. Section 5.

2.3 Inference Tasks

In probabilistic programming and statistical relational learning, the following inference tasks have been considered:

- In the $SUCC(q)$ task, a ground query q is given, and the task is to compute

$$SUCC(q) = P_s(q),$$

  the success probability of the query as specified in Equation (4).[2]

- In the $MARG(Q \mid e)$ task, a set Q of ground atoms of interest, the query atoms, and a ground query e, the evidence, are given. The task is to compute the marginal probability distribution of each atom $q \in Q$ given the evidence,

$$P_s(q|e) = \frac{P_s(q \wedge e)}{P_s(e)}.$$

  The $SUCC(q)$ task corresponds to the special case of the $MARG(Q \mid e)$ task with $Q = \{q\}$ and $e = \text{true}$ (and thus $P_s(e) = 1$).

- The $MAP(Q \mid e)$ task is to find the most likely truth-assignment q to the atoms in Q given the evidence e, that is, to compute

$$MAP(Q \mid e) = \arg\max_q P_s(Q = q|e)$$

- The $MPE(U \mid e)$ task is to find the most likely world where the given evidence query e holds. Let U be the set of all atoms in the Herbrand base that do not occur in e. Then, the task is to compute the most likely truth-assignment u to the atoms in U,

$$MPE(e) = MAP(U \mid e).$$

- In the $VIT(q)$ task, a query q is given, and the task is to find a Viterbi proof of q. Let $E(q)$ be the set of all explanations or proofs of q, that is, of all sets F' of ground probabilistic atoms for which q is true in the corresponding possible world. Then, the task is to compute

$$VIT(q) = \arg\max_{X \in E(q)} P_s(\bigwedge_{f \in X} f).$$

To illustrate, consider our initial alarm example (1) with $e = \texttt{calls(mary)}$ and $Q = \{\texttt{burglary}, \texttt{calls(john)}\}$. The worlds where the evidence holds are listed in Table 1, together with their probabilities. The answer to the MARG task is $P_s(\texttt{burglary}|\texttt{calls(mary)}) = 0.07/0.196 = 0.357$ and $P_s(\texttt{calls(john)}|\texttt{calls(mary)}) =$

[2]Non-ground queries have been considered as well, in which case the success probability corresponds to the probability that $q\theta$ is true for some grounding substitution θ.

$0.0784/0.196 = 0.4$. The answer to the MAP task is `burglary`=false, `calls(john)`=false, as its probability 0.0756/0.196 is higher than 0.028/0.196 (for true, true), 0.042/0.196 (for true, false) and 0.0504/0.196 (for false, true). The world returned by MPE is the one corresponding to the set of facts `{earthquake, hears_alarm(mary)}`. Finally, the Viterbi proof of query `calls(mary)` is $e \wedge hm$, as $0.2 \cdot 0.7 > 0.1 \cdot 0.7$ (for $b \wedge hm$).

3 Inference

We now provide an overview of existing inference approaches in probabilistic (logic) programming. As most existing work adresses the SUCC task of computing success probabilities, cf. Equation (4), we focus on this task here, and mention other tasks in passing where appropriate. For simplicity, we assume probabilistic facts as basic building blocks. Computing marginals under the distribution semantics has to take into account both *probabilistic* and *logical* aspects. We therefore distinguish between *exact* inference and approximation using either *bounds* or *sampling* on the probabilistic side, and between methods based on *forward* and *backward reasoning* and *grounding to CNF* on the logical side. Systems implementing (some of) these approaches include the ICL system AILog2[3], the PRISM system[4], the ProbLog implementations ProbLog1[5] and ProbLog2[6], and the LPAD implementations cplint[7] and PITA[8]. General statements about systems in the following refer to these six systems.

3.1 Exact Inference

As most methods for exact inference can be viewed as operating (implicitly or explicitly) on a propositional logic representation of all possible worlds that entail the query q of interest, we first note that this set of possible worlds is given by the following formula in disjunctive normal form (DNF)

$$DNF(q) = \bigvee_{\substack{F' \subseteq F \\ \exists \theta F' \cup R \models q\theta}} \left(\bigwedge_{f_i \in F'} f_i \wedge \bigwedge_{f_i \in F \setminus F'} \neg f_i \right) \tag{6}$$

and that the structure of this formula exactly mirrors that of Equation (5) defining the success probability in the case of probabilistic facts, where we replace summation by disjunction, multiplication by conjunction, and probabilities by truth values of random variables (or facts).

In our initial alarm example (1), the DNF corresponding to `calls(mary)` contains the worlds shown in Table 1, and thus is

$$\begin{aligned}(b \wedge e \wedge hm \wedge hj) \vee (b \wedge e \wedge hm \wedge \neg hj) \vee (b \wedge \neg e \wedge hm \wedge hj) \\ \vee (b \wedge \neg e \wedge hm \wedge \neg hj) \vee (\neg b \wedge e \wedge hm \wedge hj) \vee (\neg b \wedge e \wedge hm \wedge \neg hj).\end{aligned} \tag{7}$$

[3] `http://artint.info/code/ailog/ailog2.html`
[4] `http://sato-www.cs.titech.ac.jp/prism/`
[5] included in YAP Prolog, `http://www.dcc.fc.up.pt/~vsc/Yap/`
[6] `http://dtai.cs.kuleuven.be/problog/`
[7] included in YAP Prolog, `http://www.dcc.fc.up.pt/~vsc/Yap/`
[8] included in XSB Prolog, `http://xsb.sourceforge.net/`

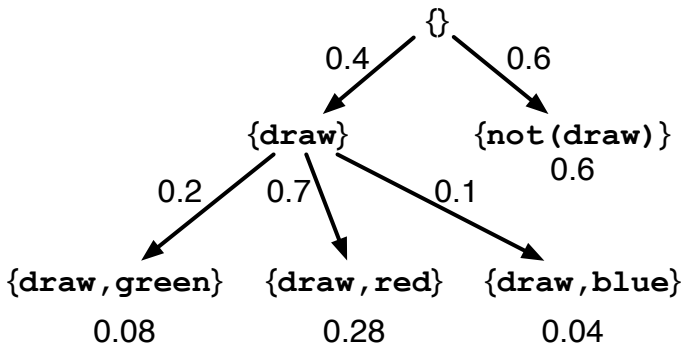

Figure 1: Forward reasoning example.

Forward Reasoning: Following the definition of the semantics of CP-logic [Vennekens et al., 2009], forward reasoning can be used to build a tree whose leaves correspond to possible worlds, on which success probabilities can be calculated. Specifically, the root of the tree is the empty set, and in each node, one step of forward reasoning is executed, creating a child for each possible outcome in the case of probabilistic facts or annotated disjunctions. For instance, consider the program

$$0.4 :: \texttt{draw}.$$
$$0.2 :: \texttt{green}; 0.7 :: \texttt{red}; 0.1 :: \texttt{blue} :- \texttt{draw}.$$

As illustrated in Figure 1, the first step using the probabilistic fact `draw` adds two children to the root, one containing `draw`, and one containing `not`(`draw`). In the latter case, the body of the AD is false and thus no further reasoning steps are possible. For the world where `draw` is true, the AD introduces three children, adding `green`, `red` and `blue`, respectively, and no further reasoning steps are possible in the resulting worlds. Thus, each path from the root to a leaf constructs one possible world, whose probability is the product of assignments made along the path. Domains for non-ground facts have to be explicitly provided to ensure termination. While this approach clearly illustrates the semantics, even in the finite case, it suffers from having to enumerate all possible worlds, and is therefore not used in practice.

Backward Reasoning: Probably the most common inference strategy in probabilistic logic programming is to collect all possible *proofs* or *explanations* of a given query using backward reasoning, represent them in a suitable data structure, and compute the probability on that structure. As discussed in Section 2.3, an explanation is a partial truth value assignment to probabilistic facts that is sufficient to prove the query via SLD-resolution. For instance, $b \wedge hm$ is the explanation for `calls(mary)` given by the derivation discussed in Section 2.1 (page 5), as it depends on `burglary` and `hears_alarm(mary)` being true, but not on any particular truth values of `earthquake` and `hears_alarm(john)`. This query has a second proof, $e \wedge hm$, obtained by using the first clause for `alarm` during backward reasoning. We can describe the set of possible worlds where `calls(mary)` is true by the disjunction of all proofs of the query,

$(b \wedge hm) \vee (e \wedge hm)$, which is more compact than the disjunction (7) explicitly listing the six possible worlds. We cannot, however, calculate the probability of this more compact DNF by simply replacing conjunction by multiplication and disjunction by addition as we did for the longer DNF above. The reason is that the two proofs are *not mutually exclusive*, that is, they can be true in the same possible world. Specifically, in our example this holds for the two worlds $b \wedge e \wedge hm \wedge hj$ and $b \wedge e \wedge hm \wedge \neg hj$, and the probability of these worlds, $0.1 \cdot 0.2 \cdot 0.7 \cdot 0.4 + 0.1 \cdot 0.2 \cdot 0.7 \cdot (1 - 0.4) = 0.014$ is exactly the difference between 0.21 as obtained by the direct sum of products $0.1 \cdot 0.7 + 0.2 \cdot 0.7$ and the true probability 0.196. This is also known as the *disjoint-sum-problem*, which is #P-complete [Valiant, 1979]. Existing languages and systems approach the problem from different angles. PHA [Poole, 1992] and PRISM [Sato and Kameya, 2001] rely on the *exclusive explanation assumption*, that is, they assume that the structure of the program guarantees mutual exclusiveness of all conjunctions in the DNF, which allows one to evaluate it as a direct sum of products (as done in the PRISM system). This assumption allows for natural modeling of many models, including e.g., probabilistic grammars and Bayesian networks, but prevents direct modeling of e.g., connection problems over uncertain graphs where each edge independently exists with a certain probability, or simple variations of Bayesian network models such as our running example. ICL [Poole, 2000] is closely related to PHA, but does not assume exclusive explanations. Poole instead suggests symbolic disjoining techniques to split explanations into mutually exclusive ones (implemented in AILog2). The ProbLog1 implementation of ProbLog [De Raedt et al., 2007, Kimmig et al., 2011a] has been the first probabilistic programming system representing DNFs as Binary Decision Diagrams (BDDs), an advanced data structure that disjoins explanations. This technique has subsequently also been adopted for ICL and LPADs in the cplint and PITA systems [Riguzzi, 2009, Riguzzi and Swift, 2011]. AILog2 and cplint also support computing conditional probabilities. Riguzzi [2013c] has introduced an approach called PITA(OPT) that automatically recognizes certain independencies that allow one to avoid the use of disjoining techniques when computing marginal probabilities. Given its focus on proofs, backward reasoning can easily be adapted to solve the VIT task of finding most likely proofs, as done in the PRISM, ProbLog1 and PITA systems.

Reduction to Weighted Model Counting: A third way to approach the logic side of inference in probabilistic logic programming has been suggested by Fierens et al. [2011, 2014], who use the propositional logic semantics of logic programming to reduce MARG inference to *weighted model counting* (WMC) and MPE inference to weighted MAX-SAT. The first step again builds a Boolean formula representing all models where the query is true, but this time, using conjunctive normal form (CNF), and associating a weight with every literal in the formula. More specifically, it grounds the parts of the logic program relevant to the query (that is, the rule groundings contributing to a proof of the query, as determined using backward reasoning), similar to what happens in answer set programming, transforms this ground program into an equivalent CNF based on the semantics of logic programming, and defines the weight function for the second step using the given probabilities. The second step can then use any existing approach to WMC or weighted MAX-SAT, such as representing

the CNF as an *sd-DNNF*, a data structure on which WMC can be performed efficiently.

For instance, the relevant ground program for `calls(mary)` in our initial alarm example (1) is

$$\begin{aligned} 0.1 :: \texttt{burglary}. \quad 0.7 :: \texttt{hears\_alarm}(\texttt{mary}). \\ 0.2 :: \texttt{earthquake}. \\ \texttt{alarm} :- \texttt{earthquake}. \\ \texttt{alarm} :- \texttt{burglary}. \\ \texttt{calls}(\texttt{mary}) :- \texttt{alarm}, \texttt{hears\_alarm}(\texttt{mary}). \end{aligned}$$

Next, the rules in the ground program are translated to equivalent formulas in propositional logic, taking into account that their head atoms can only be true if a corresponding body is true:

$$\begin{aligned} alarm &\leftrightarrow earthquake \vee burglary \\ calls(mary) &\leftrightarrow alarm \wedge hears\_alarm(mary) \end{aligned}$$

The conjunction of these formulas is then transformed into CNF as usual in propositional logic. The weight function assigns the corresponding probabilities to literals of probabilistic facts, e.g., $w(burglary) = 0.1$, $w(\neg burglary) = 0.9$, and 1.0 to all other literals, e.g., $w(calls(mary)) = w(\neg calls(mary)) = 1.0$. The weight of a model is the product of all literal weights, and the WMC of a formula the sum of weights of all its models, which exactly corresponds to the success probability. Evidence can directly be incorporated by conjoining it with the CNF. Exact MARG inference using this approach is implemented in ProbLog2.

Lifted Inference is the topic of a lot of research in statistical relational learning today [Kersting, 2012, Poole, 2003]. Lifted inference wants to realize probabilistic logic inference at the lifted, that is, non-grounded level in the same way that resolution realizes this for logical inference. The problem of lifted inference can be illustrated on the following example (cf. also Poole [2008]):

$$\begin{aligned} &\texttt{p} :: \texttt{famous}(\texttt{Y}). \\ &\texttt{popular}(\texttt{X}) :- \texttt{friends}(\texttt{X}, \texttt{Y}), \texttt{famous}(\texttt{Y}). \end{aligned}$$

In this case $P_s(\texttt{popular}(\texttt{john})) = 1 - (1-p)^m$ where m is the number of friends of `john`, that is, to determine the probability that `john` is popular, it suffices to know how many friends `john` has. We do not need to know the identities of these friends, and hence, need not ground the clauses.

Various techniques for lifted inference have been obtained over the past decade. For instance, Poole [2003] shows how variable elimination, a standard approach to probabilistic inference in graphical models, can be lifted and Van den Broeck et al. [2011] studied weighted model counting for first order probabilistic logic using a generalization of d-DNNFs for first order logic. Lifted inference techniques are – to the best of our knowledge – not yet supported by

current probabilistic logic programming language implementations, which explains why we do not provide more details in this paper. It remains a challenge for further work. A recent survey on lifted inference is provided by Kersting [2012].

3.2 Approximate Inference using Bounds

As the probability of a set of possible worlds monotonically increases if more models are added, hard lower and upper bounds on the success probability can be obtained by considering a subset or a superset of all possible worlds where a query is true. For instance, let W be the set of possible worlds where a query q holds. The success probability of q thus is the sum of the probabilities of all worlds in W. If we restrict this sum to a subset of W, we obtain a lower bound, and an upper bound if we sum over a superset of W. In our example, as `calls(mary)` is true in $b \wedge e \wedge hm \wedge hj$, but false in $b \wedge e \wedge \neg hm \wedge hj$, we have $0.1 \cdot 0.2 \cdot 0.7 \cdot 0.4 \leq P_s(\texttt{calls(mary)}) \leq 1 - (0.1 \cdot 0.2 \cdot (1 - 0.7) \cdot 0.4)$.

In practice, this approach is typically used with the DNF obtained by backward reasoning, that is, the set of proofs of the query, rather than with the possible worlds directly. This has initially been suggested for PHA by Poole [1992], and later also been adapted for ProbLog [De Raedt et al., 2007, Kimmig et al., 2008] and LPADs [Bragaglia and Riguzzi, 2011]. The idea is to maintain a set of partial derivations during backward reasoning, which allows one to, at any point, obtain a lower bound based on all complete explanations or proofs found so far, and an upper bound based on those together with all partial ones (based on the assumption that those will become proofs with probability one). For instance, $(e \wedge hm) \vee b$ provides an upper bound of 0.226 for the probability of `calls(mary)` based on the proof $e \wedge hm$ (which provides the corresponding lower bound 0.14) and the partial derivation b (which still requires to prove `hears_alarm(mary)`). Different search strategies are possible here, including e.g., iterative deepening or best first search. Lower bounds based on a fixed number of proofs have been proposed as well, either using the k explanations with highest individual probabilities [Kimmig et al., 2011a], or the k explanations chosen by a greedy procedure that maximizes the probability an explanation adds to the one of the current set [Renkens et al., 2012]. Approximate inference using bounds is available in ProbLog1, cplint, and ProbLog2.

3.3 Approximate Inference by Sampling

While probabilistic logic programming often focuses on exact inference, approximate inference by sampling is probably the most popular approach to inference in many other probabilistic languages. Sampling uses a large number of random executions or randomly generated possible worlds, from which the probability of a query is estimated as the fraction of samples where the query holds. For instance, samples can be generated by randomly choosing truth values of probabilistic facts as needed during backward reasoning, until either a proof is found or all options are exhausted [Kimmig et al., 2008, Bragaglia and Riguzzi, 2011, Riguzzi, 2013b]. Fierens et al. [2014] have used MC-SAT [Poon and Domingos, 2006] to perform approximate WMC on the CNF representing all models. Systems for languages that specify generative models, such as BLOG [Milch

et al., 2005] and distributional clauses [Gutmann et al., 2011], cf. Sec. 4.2, often use forward reasoning to generate samples. A popular approach to sampling are MCMC algorithms, which, rather than generating each sample from scratch, generate a sequence of samples by making random modifications to the previous sample based on a so-called proposal distribution. This approach has been used e.g., for the probabilistic functional programming language Church [Goodman et al., 2008], for BLOG [Arora et al., 2010], and for the probabilistic logic programming languages PRISM [Sato, 2011] and ProbLog [Moldovan et al., 2013]. ProbLog1 and cplint provide inference techniques based on backward sampling, and the PRISM system includes MCMC inference.

4 Probabilistic Programming Concepts

While probabilistic programming languages based on the distribution semantics as discussed so far are expressive enough for a wide range of models, an important part of their power is their support for additional programming concepts. Based on primitives used in a variety of probabilistic languages, we discuss a range of such concepts next, also touching upon their implications for inference.

4.1 Flexible Probabilities

A probabilistic fact with *flexible probability* is of the form $P :: atom$ where $atom$ contains the logical variable P that has to be instantiated to a probability when using the fact. The following example models drawing a red ball from an urn with R red and G green balls, where each ball is drawn with uniform probability from the urn:

```
Prob :: red(Prob).
draw_red(R,G):-   Prob is R/(R+G),
                  red(Prob).
```

The combination of flexible probabilities and Prolog code offers a powerful tool to compute probabilities on-the-fly, cf. e.g., [Poole, 2008]. Flexible probabilities have also been used in extended SLPs [Angelopoulos and Cussens, 2004], and are supported by the probabilistic logic programming systems AILog2, ProbLog1, cplint and ProbLog2. Indeed, probabilistic facts with flexible probabilities are easily supported by backward inference as long as these facts are ground on calling, but cannot directly be used with exact forward inference, as they abbreviate an infinite set of ground facts and thus would create an infinite tree of possible worlds.[9]

4.2 Distributional Clauses

Annotated disjunctions – as specified in Section 2.2 – are of limited expressivity, as they can only define distributions over a fixed, finite number of head elements. While more flexible discrete distributions can be expressed using a combination of flexible probabilities and Prolog code, this may require significant programming effort. Gutmann et al. [2010] introduce Hybrid ProbLog, an

[9] If only finitely many different instances of such a fact are relevant for any possible world of a given program, a mechanism similarly to the magic set transformation [Bancilhon et al., 1986] may circumvent this problem.

extension of ProbLog to continuous distributions, but their inference approach based on exact backward reasoning and discretization severely limits the use of such distributions. To alleviate these problems, *distributional clauses* were introduced by Gutmann et al. [2011], whom we closely follow.

A *distributional clause* is a clause of the form

$$\mathtt{h} \sim \mathcal{D} \text{ :- } \mathtt{b}_1, \ \ldots \ , \mathtt{b}_n.$$

where $\sim$ is a binary predicate used in infix notation. Similarly to annotated disjunctions, the head $(\mathtt{h} \sim \mathcal{D})\theta$ of a distributional clause is defined for a grounding substitution θ whenever $(\mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$ is true in the semantics of the logic program. Then the distributional clause defines the random variable $\mathtt{h}\theta$ as being distributed according to the associated distribution $\mathcal{D}\theta$. Possible distributions include finite discrete distributions such as a uniform distribution, discrete distributions over infinitely many values, such as a Poisson distribution, and continuous distributions such as Gaussian or Gamma distributions. The outcome of a random variable h is represented by the term $\simeq(h)$. Both random variables h and their outcome $\simeq(h)$ can be used as other terms in the program. However, the typical use of terms $\simeq(h)$ is inside comparison predicates such as `equal/2` or `lessthan/2`. In this case these predicates act in the same way as probabilistic facts in Sato's distribution semantics. Indeed, depending on the value of $\simeq(h)$ (which is determined probabilistically) they will be true or false.

Consider the following distributional clause program.

$$\begin{aligned}
\mathtt{color(B)} \sim \mathtt{discrete}((0.7:\mathtt{green}),(0.3:\mathtt{blue})) &\text{ :- } \mathtt{ball(B)}.\\
\mathtt{diameter(B,MD)} \sim \mathtt{gamma(MD1,20)} &\text{ :- } \mathtt{mean_diameter}(\simeq(\mathtt{color(B)}),\mathtt{MD}),\\
&\qquad \mathtt{MD1\ is\ MD/20}.\\
\mathtt{mean_diameter(green,15)}.&\\
\mathtt{mean_diameter(blue,25)}.&\\
\mathtt{ball(1)}.\ \mathtt{ball(2)}.\ \ldots\ \mathtt{ball(k)}.&
\end{aligned}$$

The first clause states that for every ball `B`, there is a random variable `color(B)` whose value is either `green` (with probability 0.7) or `blue` (with probability 0.3). This discrete distribution directly corresponds to the one given by the annotated disjunction `0.7 :: color(B,green); 0.3 :: color(B,blue) :- ball(B)`. The second distributional clause in the example defines a random variable `diameter(B,MD)` for each ball `B`. This random variable follows a Gamma distribution with parameters `MD/20` and 20, where the mean diameter `MD` depends on the color of the ball.

Distributional clauses are the logic programming equivalent of the mechanisms employed in statistical relational languages such as Bayesian Logic (BLOG) [Milch et al., 2005], Church [Goodman et al., 2008] and IBAL [Pfeffer, 2001], which also use programming constructs to define generative process that can define new variables in terms of existing one.

As we have seen in the example, annotated disjunctions can easily be represented as distributional clauses with finite, discrete distributions. However, distributional clauses are more expressive than annotated disjunctions (and the standard distribution semantics) as they can also represent continuous distributions.

Performing inference with distributional clauses raises some extra difficulties (see [Gutmann et al., 2011] for more details). The reason for this is that continuous distributions (such as a Gaussian or a Gamma-distribution) have uncountable domains. Typical inference with constructs such as distributional clauses will therefore resort to sampling approaches in order to avoid the need for evaluating complex integrals. It is quite natural to combine sampling for distributional clauses with forward reasoning[10], realizing a kind of generative process, though more complex strategies are also possible, cf. [Gutmann et al., 2011].

4.3 Unknown Objects

One of the key contributions of Bayesian Logic (BLOG) [Milch et al., 2005] is that it allows one to drop two common assumptions, namely the *closed world assumption* (all objects in the world are known in advance) and the *unique names assumption* (different terms denote different objects), which makes it possible to define probability distributions over outcomes with varying sets of objects. This is achieved by defining generative processes that construct possible worlds, where the existence and the properties of objects can depend on objects created earlier in the process.

As already shown by Poole [2008], such generative processes with an unknown number of objects can often be modeled using flexible probabilities and Prolog code to specify a distribution over the number of objects as done in BLOG. Distributional clauses simplify this modeling task, as they make introducing a random variable corresponding to this number straightforward. We can then use the `between/3` predicate to enumerate the objects in definitions of predicates that refer to them, cf. also [Poole, 2008]. Below, the random variable `nballs` stands for the number of balls, which is Poisson distributed with $\lambda = 6$. For each possible value $\simeq$(`nballs`), the corresponding number of balls are generated which are identified by the numbers $1, 2, \ldots, \simeq$(`nballs`).

$$\texttt{nballs} \sim \texttt{poisson}(6).$$
$$\texttt{ball}(\texttt{N}) : -\texttt{between}(1, \simeq(\texttt{nballs}), \texttt{N}).$$

4.4 Stochastic Memoization

A key concept in the probabilistic functional programming language Church [Goodman et al., 2008] is *stochastic memoization*. If a random variable in Church is memoized, subsequent calls to it simply look up the result of the first call, similarly to *tabling* in logic programming. On the other hand, for random variables that are not memoized, each reference to the variable corresponds to an independent draw of an outcome. In contrast to Church, probabilistic logic programming languages and their implementations typically do not leave this choice to the user. In ICL, ProbLog, LPADs and the basic distribution semantics as introduced in [Sato, 1995], each ground probabilistic fact directly corresponds to a random variable, i.e., within a possible world, each occurrence of such a fact has the same truth value, and the fact is thus memoized. Furthermore, the probability of the fact is taken into account once when calculating

[10] Valid distributional clause programs are required to have finite support, which ensures termination.

the probability of a proof, independently of the number of times it occurs in that proof. While early versions of PRISM [Sato, 1995, Sato and Kameya, 1997] used binary or n-ary probabilistic choices with an argument that explicitly distinguished between different calls, this argument has been made implicit later on [Sato and Kameya, 2001], meaning that the PRISM implementation never memoizes the outcome of a random variable.

The difference between the two approaches can be explained using the following example. For the AD ($\frac{1}{3}$ :: `color(green)`; $\frac{1}{3}$ :: `color(red)`; $\frac{1}{3}$ :: `color(blue)`), there are three answers to the goal `(color(X),color(Y))`, one answer `X = Y = c` for each color `c` with probability $\frac{1}{3}$, as exactly one of the facts `color(c)` is true in each possible world when memoizing color (as in ProbLog and ICL). Asking the same question when color is not memoized (as in PRISM) results in 9 possible answers with probability $\frac{1}{9}$ each. The query then – implicitly – corresponds to an ICL or ProbLog query `(color(X,id1), color(Y,id2))`, where the original AD is replaced by a non-ground variant ($\frac{1}{3}$ :: `color(green, ID)`; $\frac{1}{3}$ :: `color(red, ID)`; $\frac{1}{3}$ :: `color(blue, ID)`) and `id1` and `id2` are trial identifiers that are unique to the call.

Avoiding the memoization of probabilistic facts is necessary in order to model stochastic automata, probabilistic grammars, or stochastic logic programs [Muggleton, 1996] under the distribution semantics. There, a new rule is chosen randomly for each occurrence of the same nonterminal state/symbol/predicate within a derivation, and each such choice contributes to the probability of the derivation. The rules for a nonterminal thus form a family of independent identically distributed random variables, and each choice is automatically associated with one variable from this family.

Consider the following stochastic logic program. It is in fact a fragment of a stochastic definite clause grammar; the rules essentially encode the probabilistic context free grammar rules defining $0.3 : vp \rightarrow verb$, $0.5 : vp \rightarrow verb, np$ and $0.2 : vp \rightarrow verb, pp$. There are three rules for the non-terminal vp and each of them is chosen with an associated probability. Furthermore, the sum of the probabilities for these rules equals 1.

```
0.3 : vp(H,T) :- verb(H,T).
0.5 : vp(H,T) :- verb(H,H1),np(H1,T).
0.2 : vp(H,T) :- verb(H,H1),pp(H1,T).
```

This type of stochastic grammar can easily be simulated in the distribution semantics using one *dememoized* AD (or switch) for each non-terminal, a rule calling the AD to make the selection, and a set of rules linking the selection to

the SLP rules:[11]

```
mmemoize 0.3 :: vp_sel(rule1); 0.5 :: vp_sel(rule2); 0.2 :: vp_sel(rule3).
vp(H,T) :- vp_sel(Rule), vp_rule(Rule,H,T).
vp_rule(rule1,H,T) :- verb(H,T).
vp_rule(rule2,H,T) :- verb(H,H1), np(H1,T).
vp_rule(rule3,H,T) :- verb(H,H1), pp(H1,T).
```

All inference approaches discussed here naturally support stochastic memoization; this includes the ones implemented in AILog2, ProbLog1, ProbLog2, cplint and PITA. The PRISM system uses exact inference based on backward reasoning in the setting without stochastic memoization. In principle, stochastic memoization can be disabled in backward reasoning by automatically adding a unique identifier to each occurrence of the same random variable. However, for techniques that build propositional representations different from mutually exclusive DNFs (such as the DNFs of BDD-based methods and the CNFs when reducing to WMC), care is needed to ensure that these identifiers are correctly shared among different explanations when manipulating these formulas. Backward sampling can easily deal with both memoized and mememoized random variables. As only one possible world is considered at any point, each repeated occurrence of the same mememoized variable is simply sampled independently, whereas the first result sampled within the current world is reused for memoized ones. Forward sampling cannot be used without stochastic memoization, as it is unclear up front how many instances are needed. MCMC methods have been developed both for ProbLog (with memoization) and PRISM (without memoization).

4.5 Constraints

In knowledge representation, answer set programming and databases, it is common to allow the user to specify constraints on the possible models of a theory. In knowledge representation, one sometimes distinguishes inductive definitions (such as the definite clauses used in logic programming) from constraints. The former are used to define predicates, the latter impose constraints on possible worlds. While the use of constraints is still uncommon in probabilistic logic programming[12] it is conceptually easy to accommodate this when working with the distribution semantics, cf. Fierens et al. [2012]. While such constraints can in principle be any first-order logic formula, we will employ clausal constraints here.

A *clausal constraint* is an expression of the form

$$\mathtt{h}_1;\ \ldots\ ;\mathtt{h}_N :\!-\ \mathtt{b}_1,\ \ldots\ ,\mathtt{b}_M.$$

where the h_i and b_j are literals. The constraint specifies that whenever $(b_1 \ldots b_M)\theta$ is true for a substitution θ grounding the clause at least one of the $h_i\theta$ must also

[11] The `mmemoize` keyword is used for clarity here; it is not supported by existing systems.

[12] Hard and soft constraints are used in Markov Logic [Richardson and Domingos, 2006], but Markov Logic does not support inductive definitions as this requires a least Herbrand semantics, cf. Fierens et al. [2012].

be true. All worlds in which a constraint is violated become impossible, that is, their probability becomes 0. Constraints are very useful for specifying complex properties that possible worlds must satisfy.

To illustrate constraints, reconsider the alarm example and assume that it models a situation in the 1930s where there is only one phone available in the neighborhood implying that at most one person can call. This could be represented by the constraint

$$\texttt{X} = \texttt{Y} \texttt{ :- } \texttt{calls(X), calls(Y).}$$

Imposing this constraint would exclude all worlds in which both Mary and John hear the alarm and call. The total probability mass for such worlds is $0.4 \cdot 0.8 = 0.32$. By excluding these worlds, one looses probability mass and thus has to normalize the probabilities of the remaining possible worlds. For instance, the possible world corresponding to the truth value assignment `burglary`=true, `earthquake`=false, `hears_alarm(mary)`=true, `hears_alarm(john)`=false yielded a probability mass of $0.1 \cdot (1-0.2) \cdot 0.7 \cdot (1-0.6) = 0.0336$ without constraints. Now, when enforcing the constraint, one obtains $0.0336/(1-0.32)$. Thus the semantics of constraints correspond to computing conditional probabilities where one conditions on the constraints being satisfied.

Handling constraints during inference has not been a focus of inference in probabilistic logic programming, and – to the best of our knowledge – no current system provides explicit support for constraints.

4.6 Negation as Failure

So far, we have only considered probabilistic programs using definite clauses, that is, programs that only use positive literals in clause bodies, as those are guaranteed to have a unique model for any truth value assignment to basic probabilistic events. It is however possible to adopt Prolog's *negation as failure* on ground literals under the distribution semantics, as long as all truth values of derived atoms are still uniquely determined by those of the basic facts, cf., e.g., [Poole, 2000, Sato et al., 2005, Kimmig et al., 2009, Riguzzi, 2009, Fierens et al., 2014]. Then, in each possible world, any ground query `q` either succeeds or fails, and its negation `not(q)` succeeds in exactly those worlds where `q` fails. Thus, the probability of a ground query `not(q)` is the sum of the probabilities of all possible worlds that do *not* entail `q`. Consider the following variant of our alarm example, where people also call if there is no alarm, but they have gossip to share:

$$\begin{aligned}
0.1 &:: \texttt{burglary}. & 0.7 &:: \texttt{hears_alarm(mary)}.\\
0.2 &:: \texttt{earthquake}. & 0.4 &:: \texttt{hears_alarm(john)}.\\
0.3 &:: \texttt{has_gossip(mary)}. & 0.6 &:: \texttt{has_gossip(john)}.\\
\texttt{alarm} &\texttt{ :- earthquake}.\\
\texttt{alarm} &\texttt{ :- burglary}.\\
\texttt{calls(X)} &\texttt{ :- alarm, hears_alarm(X)}.\\
\texttt{calls(X)} &\texttt{ :- not(alarm), has_gossip(X)}.\\
\texttt{call} &\texttt{ :- calls(X)}.
\end{aligned}$$

The new rule for `calls(X)` can only possibly apply in worlds where `not(alarm)` succeeds, that is, `alarm` fails, which are exactly those containing neither `burglary` nor `earthquake`. Using gm as shorthand for `has_gossip(mary)`= true, we obtain the additional explanation $\neg e \wedge \neg b \wedge gm$ for `calls(mary)`. Thus, in the presence of negation, explanations no longer correspond to sets of probabilistic facts as in the case of definite clause programs, but to sets of positive and negative literals for probabilistic facts. While `not(alarm)` has a single explanation in this simple example, in general, explanations for negative literals can be much more complex, as they have to falsify every possible explanation of the corresponding positive literal by flipping the truth value of at least one probabilistic fact included in the explanation.

Negation as failure can be handled in forward and backward reasoning both for exact inference and for sampling, though forward reasoning has to ensure to proceed in the right order. Exact inference with backward reasoning often benefits from tabling. Negation as failure complicates approximate inference using bounds, as explanations for failing goals have to be considered. AILog2, ProbLog1, ProbLog2, cplint and PITA all support negation as failure in their exact and sampling based approaches. The PRISM system follows the approach proposed by Sato et al. [2005] and compiles negation into a definite clause program with unification constraints. Current MCMC approaches in probabilistic logic programming do not support negation beyond that of probabilistic facts.

4.7 Second Order Predicates

When modeling relational domains, it is often convenient to reason over sets of objects that fullfil certain conditions, for instance, to aggregate certain values over them. In logic programming, this is supported by second order predicates such as `findall/3`, which collects all answer substitutions for a given query in a list. In the following example, the query `sum(S)` will first collect all arguments of `f/1` into a list and then sum the values using predicate `sum_list/2`, thus returning `S=3`.

```
f(1).
f(2).
sum(Sum) :- findall(X,f(X),L),sum_list(L,Sum).
```

Note that in Prolog, the list returned by `findall/3` is unique. Under the distribution semantics, however, this list will be different depending on which possible world is considered. To illustrate this, we replace the definition of `f/1` in our example with probabilistic facts:

```
0.1 :: f(1).
0.2 :: f(2).
sum(Sum) :- findall(X,f(X),L),sum_list(L,Sum).
```

We now have four sets of facts – `{f(1),f(2)}`, `{f(1)}`, `{f(2)}`, and `{}` – leading to the four possible worlds `{f(1),f(2),sum(3)}`, `{f(1),sum(1)}`, `{f(2),sum(2)}`, and `{sum(0)}`, as the answer list `L` is different in each case.

This behavior of second order predicates in the probabilistic setting can pose a challenge to inference. In principle, all inference approaches could deal with

second order predicates. However, exact approaches would suffer from a blow-up, as they have to consider all possible lists of elements – and thus all possible worlds – explicitly, whereas in sampling, each sample only considers one such list. As far as we know, the only system with some support for second order predicates is cplint, which allows `bagof` and `setof` with one of its backward reasoning modules [Riguzzi, 2013a].

4.8 Meta-Calls

One of the distinct features of programming languages such as Prolog and Lisp is the possibility to use programs as objects within programs, which enables meta-level programming. For their probabilistic extensions, this means reasoning about the probabilities of queries within a probabilistic program, a concept that is central to the probabilistic programming language Church, which builds upon a Lisp dialect [Goodman et al., 2008], and has also been considered with ProbLog [Mantadelis and Janssens, 2011]. Possible uses of such a feature include filtering of proofs based on the probability of subqueries, or the dynamic definition of probabilities using queries, e.g., to implement simple forms of combining rules as in the following example, where `max_true(G1,G2)` succeeds with the success probability of the more likely argument.

```
P :: p(P).
max_true(G1,G2) :- prob(G1,P1),prob(G2,P2),max(P1,P2,P),p(P).
% rest of program (omitted)
```

In this section, we will use `prob(Goal,Prob)` to refer to an atom returning the success probability `Prob` of goal `Goal`, that is, implementing Equation (4). Note that such atoms are independent queries, that is, they do not share truth values of probabilistic facts with other atoms occurring in a derivation they are part of. Finally, if the second argument is a free variable upon calling, the success probability of `prob(goal,Prob)` is 1. For the sake of simplicity, we will assume here that the second argument will always be free upon calling.[13]

We extend the example above with the following program.

```
0.5 :: a.   0.7 :: b.   0.2 :: c.
d :- a,not(b).
e :- b,c.
```

Querying for `max_true(d,e)` using backward reasoning will execute two calls to `prob/2` in sequence: `prob(d,P1)` and `prob(e,P2)`. Note that if multiple calls to `prob/2` atoms occur in a proof, they are independent, i.e., even if they use the same probabilistic facts, those will (implicitly) correspond to different copies of the corresponding random variables local to that specific `prob/2` call. Put differently, `prob/2` *encapsulates* part of our possible worlds. In the example, `b` is thus a different random variable in `prob(d,P1)` and `prob(e,P2)`. The reason for this encapsulation is twofold: first, the probability of a goal is not influenced by calculating the probability of another (or even the same) event before, and second, as `prob/2` summarizes a set of possible worlds, the value of

[13] This is not a restriction, as `prob(Goal,c)` is equivalent to `prob(Goal,P),P=c`.

a random variable cannot be made visible to the outside world, as it may be different in different internal worlds. Indeed, in our example, `b` needs to be false to prove `d`, but true to prove `e`, so using the same random variable would force the top level query to be unprovable. We thus obtain a kind of hierarchically organized world: some probabilistic facts are used in the top level query, others are encapsulated in `prob/2` atoms, whose queries might in turn rely on both directly called probabilistic facts and further calls to `prob/2`. In our example, `prob(d,P1)` uses random variables corresponding to probabilistic facts `a` and `b`, returning $\texttt{P1} = 0.5 \cdot (1 - 0.7) = 0.15$, `prob(e,P2)` uses random variables corresponding to probabilistic facts `b` and `c`, returning $\texttt{P2} = 0.7 \cdot 0.2 = 0.14$, and the top level query `max_true(d,e)` uses probabilistic fact `p(0.15)` and has probability $P(\texttt{more\_likely\_is\_true}(\texttt{d},\texttt{e})) = 0.15$.

The probability of a derivation is determined by the probabilities of the probabilistic facts it uses outside all `prob/2` calls. Those facts define the possible worlds from the point of view of the top level query. In those worlds, the random variables of the encapsulated parts are hidden, as they have been aggregated by `prob/2`. Returning to our example and abstracting from the concrete remainder of the program, we observe that for any given pair of goals `g1,g2` and suitable program defining those goals, `max_true(g1,g2)` has exactly one proof: the first two body atoms always succeed and return the probabilities of the goals, the third atom deterministically finds the maximum m of the two probabilities, and the proof finally uses a single random variable `p(m)` with probability m. Thus, the query indeed succeeds with the probability of the more likely goal.

Another example for the use of `prob/2` is filtering goals based on their probability:

$$\texttt{almost\_always\_false}(\texttt{G}) :- \texttt{prob}(\texttt{G},\texttt{P}), \texttt{P} < 0.00001.$$
$$\% \texttt{ rest of program (omitted)}$$

Note that in contrast to the previous example, this is a purely logical decision, that is, the success probability will be either 0 or 1 depending on the goal G.

To summarize, using meta-calls to turn probabilities into usable objects in probabilistic logic programming is slightly different from the other probabilistic programming concepts considered in this paper: it requires a notion of encapsulation or hierarchical world structure and cannot be interpreted directly on the level of individual possible worlds for the entire program.

Mantadelis and Janssens [2011] introduce MetaProbLog[14], a prototype implementation for ProbLog supporting nested meta-calls based on exact backward inference. As they discuss, meta-calls can be supported by any inference mechanism that can be suspended to perform inference for the query inside the meta-call. Such suspending is natural in backward reasoning, where the proof of a subgoal becomes a call to inference rather than a continuation of backward reasoning. With forward reasoning, such non-ground `prob(goal,P)` goals raise the same issues as other non-ground facts. Meta-calls of the form `prob(goal,P)` compute the grounding of `P` as the goal's probability, and using approximate inference to compute the latter will thus influence the grounding of such a fact, and therefore potentially also the consequences of this fact. This may affect

[14] `http://people.cs.kuleuven.be/~theofrastos.mantadelis/tools/metaproblog.tar.gz`, also supports flexible probabilities, stochastic memoization, and negation as failure

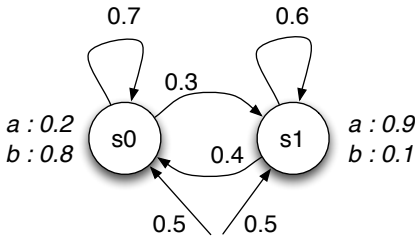

Figure 2: Example HMM

the result of inference in unexpected ways, and it is thus unclear in how far approximation approaches are suitable for meta-calls. Goodman et al. [2008] state that supporting meta-calls (or nested queries) in MCMC inference in Church is expected to be straightforward, but do not provide details. Meta-calls are not supported in AILog2, PRISM, ProbLog1, ProbLog2, cplint and PITA.

4.9 Time and Dynamics

Among the most popular probabilistic models are those that deal with dynamics and time such as Hidden Markov Models (HMMs) and Dynamic Bayesian Networks. Dynamic models have received quite some attention within probabilistic logic programming. They can naturally be represented using logic programs through the addition of an extra "time" argument to each of the predicates. We illustrate this by giving two encodings of the Hidden Markov Model shown in Figure 2, where we restrict sequences to a given length (10 in the example). Following Vennekens et al. [2004], this model can be written as a set of annotated disjunctions:

$$\texttt{length}(10).$$

$$0.5 :: \texttt{state}(\texttt{s0},0); 0.5 :: \texttt{state}(\texttt{s1},0).$$

$$0.7 :: \texttt{state}(\texttt{s0},\texttt{T1});\ 0.3 :: \texttt{state}(\texttt{s1},\texttt{T1}) {:-} \texttt{state}(\texttt{s0},\texttt{T}), \texttt{length}(\texttt{L}), \texttt{T} < \texttt{L}, \texttt{T1 is T} + 1.$$

$$0.4 :: \texttt{state}(\texttt{s0},\texttt{T1});\ 0.6 :: \texttt{state}(\texttt{s1},\texttt{T1}) {:-} \texttt{state}(\texttt{s1},\texttt{T}), \texttt{length}(\texttt{L}), \texttt{T} < \texttt{L}, \texttt{T1 is T} + 1.$$

$$0.2 :: \texttt{out}(\texttt{a},\texttt{T});\ 0.8 :: \texttt{out}(\texttt{b},\texttt{T}) {:-} \texttt{state}(\texttt{s0},\texttt{T}).$$

$$0.9 :: \texttt{out}(\texttt{a},\texttt{T});\ 0.1 :: \texttt{out}(\texttt{b},\texttt{T}) {:-} \texttt{state}(\texttt{s1},\texttt{T}).$$

Alternatively, following Sato and Kameya [1997], but writing PRISM's multi-valued switches as unconditional annotated disjunctions[15], the model can be

[15]In this example, the program structure causes the time argument to act as a unique identifier for different calls to the same AD, thus making memoized ADs and dememoized switches equivalent.

Figure 3: Example graph illustrating generalized labels: Boolean case (left), shortest path (right).

written as follows:

```
0.2 :: output(s0, a, T) ; 0.8 :: output(s0, b, T).
0.9 :: output(s1, a, T) ; 0.1 :: output(s1, b, T).

0.5 :: init(s0) ; 0.5 :: init(s1).

0.7 :: trans(s0, s0, T) ; 0.3 :: trans(s0, s1, T).
0.4 :: trans(s1, s0, T) ; 0.6 :: trans(s1, s1, T).

length(10).

hmm(List) :- init(S), hmm(1, S, List).

% last time T :
hmm(T, S, [Obs]) :- length(T), output(S, Obs, T).

% earlier time T : output Obs in state S, transit from S to Next
hmm(T, S, [Obs|R]) :- length(L), T < L,
                      output(S, Obs, T), trans(S, Next, T),
                      T1 is T + 1, hmm(T1, Next, R).
```

Forward and backward sampling naturally deal with a time argument (provided time is bounded in the case of forward reasoning). Naively using such a time argument with exact inference results in exponential running times (in the number of time steps), though this can often be avoided using dynamic programming approaches and principles, as shown by the PRISM system, which achieves the same time complexity for HMMs as corresponding special-purpose algorithms [Sato and Kameya, 2001].

Other approaches that have devoted special attention to modeling and inference for dynamics include Logical HMMs [Kersting et al., 2006], a language for modeling HMMs with structured states, CPT-L [Thon et al., 2011], a dynamic version of CP-logic, and the work on a particle filter for dynamic distributional clauses [Nitti et al., 2013].

4.10 Generalized Labels

As we have seen in Section 3, computing success probabilities in probabilistic logic programming is closely related to evaluating the truth value of a logical formula. Weighted logic programming languages such as Dyna [Eisner et al.,

2005][16] and aProbLog [Kimmig et al., 2011b] take this observation a step further and replace probabilities (or Boolean truth values) by elements from an arbitrary semiring and corresponding combination operators.[17]

More specifically, Dyna assigns labels to ground facts in a logic program and computes weights of atoms in the heads of clauses as follows: conjunction (`,`) in clause bodies is replaced by semiring multiplication $\otimes$, that is, the weight of a body is the $\otimes$-product of the weights of its atoms, and if multiple clauses share the same head atom, this atom's weight is the $\oplus$-sum of the corresponding bodies, that is, $:-$ is replaced by semiring addition $\oplus$. We illustrate the idea with a logic program defining reachability in a directed graph adapted from Cohen et al. [2008]:

$$\texttt{reachable(S)} :- \texttt{initial(S)}.$$
$$\texttt{reachable(S)} :- \texttt{reachable(R)}, \texttt{edge(R, S)}.$$

which in Dyna is interpreted as a system of (recursive) semiring equations

$$reachable(S) \oplus = initial(S).$$
$$reachable(S) \oplus = reachable(R) \otimes edge(R, S).$$

To get the usual logic programming semantics, we can combine this program with facts labeled with values from the Boolean semiring (with $\otimes = \wedge$ and $\oplus = \vee$):

$$\texttt{initial(a)} = \texttt{T}$$
$$\texttt{edge(a, b)} = \texttt{T} \quad \texttt{edge(a, d)} = \texttt{T} \quad \texttt{edge(b, c)} = \texttt{T} \quad \texttt{edge(d, b)} = \texttt{T} \quad \texttt{edge(d, c)} = \texttt{T}$$

which means that the weights of `reachable` atoms are computed as follows:

$$reachable(a) = initial(a) = T$$
$$reachable(d) = reachable(a) \wedge edge(a, d) = T$$
$$reachable(b) = reachable(a) \wedge edge(a, b) \vee reachable(d) \wedge edge(d, b) = T$$
$$reachable(c) = reachable(b) \wedge edge(b, c) \vee reachable(d) \wedge edge(d, c) = T$$

Alternatively, one can label facts with non-negative numbers denoting costs and use $\otimes = +$ and $\oplus = \min$ to describe single-source shortest paths:

$$\texttt{initial(a)} = \texttt{0}$$
$$\texttt{edge(a, b)} = \texttt{7} \quad \texttt{edge(a, d)} = \texttt{5} \quad \texttt{edge(b, c)} = \texttt{13} \quad \texttt{edge(d, b)} = \texttt{4} \quad \texttt{edge(d, c)} = \texttt{9}$$

resulting in evaluation

$$reachable(a) = initial(a) = 0$$
$$reachable(d) = reachable(a) + edge(a, d) = 5$$
$$reachable(b) = \min(reachable(a) + edge(a, b), reachable(d) + edge(d, b)) = 7$$
$$reachable(c) = \min(reachable(b) + edge(b, c), reachable(d) + edge(d, c)) = 14$$

[16] Dyna is currently being extended into a more general language [Eisner and Filardo, 2011], but we consider the initial version here, as that one is more closely related to the probabilistic programming languages we discuss.

[17] A *semiring* is a structure $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$, where *addition* $\oplus$ is an associative and commutative binary operation over the set $\mathcal{A}$, *multiplication* $\otimes$ is an associative binary operation over the set $\mathcal{A}$, $\otimes$ distributes over $\oplus$, $e^{\oplus} \in \mathcal{A}$ is the neutral element of $\oplus$, i.e., for all $a \in \mathcal{A}$, $a \oplus e^{\oplus} = a$, $e^{\otimes} \in \mathcal{A}$ is the neutral element of $\otimes$, i.e., for all $a \in \mathcal{A}$, $a \otimes e^{\otimes} = a$, and for all $a \in \mathcal{A}$, $e^{\oplus} \otimes a = a \otimes e^{\oplus} = e^{\oplus}$. In a *commutative semiring*, $\otimes$ is commutative as well.

That is, the values of `reachable` atoms now correspond to the length of the shortest path rather than the existence of a path.

Given its origins in natural language processing, Dyna is closely related to PRISM in two aspects. First, it does not memoize labeled facts, but takes into account their weights each time they appear in a derivation, generalizing how each use of a rule in a probabilistic grammar contributes to a derivation. Second, again as in probabilistic grammars, it sums the weights of all derivations, but in contrast to PRISM or grammars does not require them to be mutually exclusive to do so.

The inference algorithm of basic Dyna as given by Eisner et al. [2005][18] computes weights by forward reasoning, keeping intermediate results in an agenda and updating them until a fixpoint is reached, though other execution strategies could be used as well, cf. [Eisner and Filardo, 2011].

As Dyna, aProbLog [Kimmig et al., 2011b] replaces probabilistic facts by semiring-labeled facts, with the key difference that it bases the labels of derived facts on the labels of their models rather than those of their derivations, which requires semirings to be commutative. It thus directly generalizes the success probability (5) and the possible world DNF (6). ProbLog inference algorithms based on BDDs or sd-DNNFs can be directly adapted to aProbLog.[19]

Rather than replacing probabilities with semiring labels, one can also combine them with utilities or costs, and use the resulting language for decision making under uncertainty, as done in DTProbLog [Van den Broeck et al., 2010].[20]

5 Knowledge-Based Model Construction

So far, we have focused on probabilistic logic languages with strong roots in logic, where the key concepts of logic and probability are unified, that is, a random variable corresponds to a ground fact (or sometimes a ground term, as in distributional clauses), and standard logic programs are used to specify knowledge that can be derived from these facts. In this section, we discuss a second important group of probabilistic logic languages with strong roots in probabilistic graphical models, such as Bayesian or Markov networks. These formalisms typically use logic as a templating language for graphical models in relational domains, and thus take a quite different approach to combine logic and probabilities, also known as *knowledge-based model construction* (KBMC). Important representatives of this stream of research include PLPs [Haddawy, 1994], PRMs [Getoor et al., 2007], BLPs [Kersting and De Raedt, 2008], LBNs [Fierens et al., 2005], RBNs [Jaeger, 1997], CLP($\mathcal{BN}$) [Santos Costa et al., 2008], chain logic [Hommersom et al., 2009], Markov Logic [Richardson and Domingos, 2006], and PSL [Broecheler et al., 2010].

In the following, we relate the key concepts underlying the knowledge-based model construction approach to those discussed in the rest of this paper. We again focus on languages based on logic programming, such as PLPs, BLPs, LBNs, chain logic, and CLP($\mathcal{BN}$), but mostly abstract from the specific lan-

[18] Implementation available at `http://dyna.org/`

[19] A prototype implementation of aProbLog is included in ProbLog1, cf. Footnote 5.

[20] An implementation of DTProbLog is included in ProbLog1 and ProbLog2, cf. Footnotes 5 and 6.

guage. These representation languages are typically designed so that implication in logic (":−") corresponds to the direct influence relation in Bayesian networks. The logical knowledge base is then used to construct a Bayesian network. So inference proceeds in two steps: the logical step, in which one constructs the network, and the probabilistic step, in which one performs probabilistic inference on the resulting network. We first discuss modeling Bayesian networks and their relational counterpart in the context of the distribution semantics, and then focus on CLP($\mathcal{BN}$) as an example of a KBMC approach whose primitives clearly expose the separation between model construction via logic programming and probabilistic inference on the propositional model.

5.1 Bayesian Networks and Conditional Probability Tables

A Bayesian network (BN) defines a joint probability distribution over a set of random variables $\mathcal{V} = \{V_1, \ldots, V_m\}$ by factoring it into a product of conditional probability distributions, one for each variable V_i given its parents $par(V_i) \subseteq \mathcal{V}$. The parent relation is given by an acyclic directed graph (cf. Figure 4), where the random variables are the nodes and an edge $V_i \to V_j$ indicates that V_i is a parent of V_j. The conditional probability distributions are typically specified as *conditional probability tables* (CPTs), which form the key probabilistic concept of BNs. For instance, the CPT on the left of Figure 4 specifies that the random variable `sprinkler` takes value true with probability 0.1 (and false with 0.9) if its parent `cloudy` is true, and with probability 0.5 if `cloudy` is false. Formally, a CPT contains a row for each possible assignment $x_1, \ldots, x_n$ to the parent variables $X_1, \ldots, X_n$ specifying the distribution $P(X|x_1, \ldots, x_n)$. As has been shown earlier, e.g., by Poole [1993] and Vennekens et al. [2004], any Bayesian network can be modeled in languages based on the distribution semantics by representing every row in a CPT as an annotated disjunction

$$p_1 :: X(w_1); \cdots ; p_k :: X(w_k) :- X_1(v_1), \cdots , X_n(v_n)$$

where $X(v)$ is true when v is the value of X. The body of this AD is true if the parent nodes have the values specified in the corresponding row of the CPT, in which case the AD chooses a value for the child from the corresponding distribution. As an example, consider the sprinkler network shown in Figure 4. The CPT for the root node `cloudy` corresponds to an AD with empty body

$$0.5 :: \texttt{cloudy}(\texttt{t}); 0.5 :: \texttt{cloudy}(\texttt{f}).$$

whereas the CPTs for `sprinkler` and `rain` require the state of their parent node `cloudy` to be present in the body of the ADs

$$\begin{aligned}
0.1 :: \texttt{sprinkler}(\texttt{t}); 0.9 :: \texttt{sprinkler}(\texttt{f}) &:- \texttt{cloudy}(\texttt{t}).\\
0.5 :: \texttt{sprinkler}(\texttt{t}); 0.5 :: \texttt{sprinkler}(\texttt{f}) &:- \texttt{cloudy}(\texttt{f}).\\
0.8 :: \texttt{rain}(\texttt{t}); 0.2 :: \texttt{rain}(\texttt{f}) &:- \texttt{cloudy}(\texttt{t}).\\
0.2 :: \texttt{rain}(\texttt{t}); 0.8 :: \texttt{rain}(\texttt{f}) &:- \texttt{cloudy}(\texttt{f}).
\end{aligned}$$

The translation for the CPT of `grass_wet` is analogous.

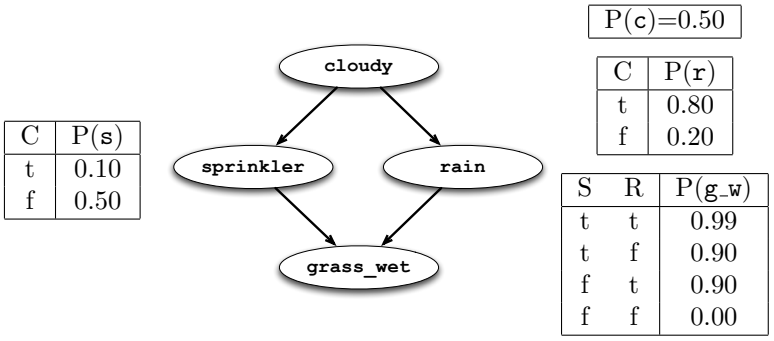

Figure 4: The sprinkler network is a Bayesian network modeling an environment where both the sprinkler and the rain can cause the grass getting wet [Russell and Norvig, 2003].

5.2 Relational Dependencies

Statistical relational learning formalisms such as BLPs, PLPs, LBNs and CLP($\mathcal{BN}$) essentially replace the specific random variables in the CPTs of Bayesian networks by logically defined random variable templates, commonly referred to as *parameterized* random variables or par-RVs for short [Poole, 2003], though the actual syntax amongst these systems differs significantly. We here use annotated disjunctions to illustrate the key idea. For instance, in a propositional setting, the following annotated disjunctions express that a specific student's grade in a specific course probabilistically depends on whether he has read the corresponding textbook or not:

$$0.6 :: \texttt{grade(high)}; 0.4 :: \texttt{grade(low)} :- \texttt{reads(true)}.$$
$$0.1 :: \texttt{grade(high)}; 0.9 :: \texttt{grade(low)} :- \texttt{reads(false)}.$$

Using logical variables, this dependency can directly be expressed for many students, courses, and books:

$$0.6 :: \texttt{grade(S,C,high)}; 0.4 :: \texttt{grade(S,C,low)} :- \texttt{book(C,B)}, \texttt{reads(S,B)}.$$
$$0.1 :: \texttt{grade(S,C,high)}; 0.9 :: \texttt{grade(S,C,low)} :- \texttt{book(C,B)}, \texttt{not(reads(S,B))}.$$

More concretely, the annotated disjunctions express that $P(grade(S,C) = high) = 0.6$ if the student has read the book of the course and $P(grade(S,C) = high) = 0.1$ otherwise. Thus the predicate `grade` depends on `book/2` and `reads/2`. The dependency holds for all instantiations of the rule, that is, it acts as a template for all persons, courses, and books. This is what knowledge-based model construction approaches all share: the logic acts as a template to generate dependencies (here CPTs) in the graphical model. This also introduces a complication that is not encountered in propositional Bayesian networks or their translation to annotated disjunctions. To illustrate this, let us assume the predicate `book/2` is deterministic and known. Then the propositional case arises when for each course there is exactly one book. The annotated disjunctions then effectively encode the conditional probability table $P(Grade|Reads)$. However,

if there are multiple books, say two, for one course, then the above template would specify two CPTs: one for the first book, $P(Grade|Reads1)$, and one for the second, $P(Grade|Reads2)$. In Bayesian networks, these CPTs need to be combined and there are essentially two ways for realizing this.

The first is to use a so-called *combining rule*, that is, a function that maps these CPTs into a single CPT of the form $P(Grade|Reads1, Reads2)$. The most popular combining rule is noisy-or, for which $P(Grade = high|Reads_1, ..., Reads_n) = 1 - \prod_{i=1}^{n}(1 - P(Grade = high|Reads_i = \text{true}))$ where n is the number of books for the course. Using annotated disjunctions, this combining rule is obtained automatically, cf. Section 2.2. In the statistical relational learning literature, this approach is followed for instance in RBNs and BLPs, and several other combining rules exist, cf., e.g., [Jaeger, 1997, Kersting and De Raedt, 2008, Natarajan et al., 2005]. While combining rules are an important concept in KBMC, using them in their general form under the distribution semantics requires one to change the underlying logic, which is non-trivial. Hommersom and Lucas [2011] introduce an approach that models these interactions by combining the distribution semantics with default logic. Alternatively, one could use meta-calls, cf. Section 4.8.

The second way of dealing with the two distributions uses aggregation. In this way, the random variable upon which one conditions grade is the number of books the person read, rather than the reading of the individual books. This approach is taken for instance in PRMs and CLP($\mathcal{BN}$). In the context of the distribution semantics, aggregation can be realized within the logic program using second order predicates, cf. Section 4.7. For instance, the following program makes a distinction between reading more than two, two, one, or none of the books:

```
0.9 :: grade(S,C,high); 0.1 :: grade(S,C,low) :- nofbooksread(S,C,N), N > 2.
0.8 :: grade(S,C,high); 0.2 :: grade(S,C,low) :- nofbooksread(S,C,2).
0.6 :: grade(S,C,high); 0.4 :: grade(S,C,low) :- nofbooksread(S,C,1).
0.1 :: grade(S,C,high); 0.9 :: grade(S,C,low) :- nofbooksread(S,C,0).
nofbooksread(S,C,N) :- findall(B,(book(C,B),reads(S,B)),List),length(List,N).
```

5.3 Example: CLP($\mathcal{BN}$)

We now discuss CLP($\mathcal{BN}$) [Santos Costa et al., 2008] in more detail, as it clearly exposes the separation between model construction via logic programming and probabilistic inference on the propositional model in KBMC. CLP($\mathcal{BN}$) is embedded in Prolog[21] and uses constraint programming principles to construct a Bayesian network via logical inference. Syntactically, CLP($\mathcal{BN}$) extends logic programming with *constraint atoms* that (a) define random variables together with their CPTs and (b) establish constraints linking these random variables to logical variables used in the logic program. The answer to a query in CLP($\mathcal{BN}$) is the marginal distribution of the query variables, conditioned on evidence if available. The first phase of inference in CLP($\mathcal{BN}$) uses backward reasoning in the logic program to collect all relevant constraints in a constraint store, the

[21] implementation included in YAP Prolog, `http://www.dcc.fc.up.pt/~vsc/Yap/`

second phase computes the marginals in the Bayesian network defined by these constraints. Conditioning on evidence is straightforward, as it only requires to add the corresponding constraints to the store.[22]

Specifically, a CLP($\mathcal{BN}$) clause (in canonical form) is either a standard Prolog clause, or has the following structure:

$$\mathtt{h}(\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{V}) :- \mathtt{body},\{\mathtt{V} = \mathtt{sk}(\mathtt{C}_1,\ldots,\mathtt{C}_t)\ \mathtt{with\ CPT}\}.$$

Here, `body` is a possibly empty conjunction of logical atoms, and the part in curly braces is a constraint atom. $\mathtt{sk}(\mathtt{C}_1,\ldots,\mathtt{C}_t)$ is a Skolem term not occurring in any other clause of the program (whose arguments $\mathtt{C}_i$ are given via the input variables $\mathtt{A}_j$ and the logical `body`), and CPT is a term of the form $\mathtt{p}(\mathtt{Values},\mathtt{Table},\mathtt{Parents})$, where `Values` is a list of possible values for $\mathtt{sk}(\mathtt{C}_1,\ldots,\mathtt{C}_t)$, `Parents` is a list of logical variables specifying the parent nodes, and `Table` the probability table given as a list of probabilities, where the order of entries corresponds to the valuations obtained by backtracking over the parents' values in the order given in the corresponding definitions. This CPT term can be given either directly or via the use of logical variables and unification.

We first illustrate this for the propositional case, using the following model[23] of the sprinkler Bayesian network as given in Figure 4:[24]

```
cloudy(C) :-
        { C = cloudy  with p([f,t],[0.5,0.5],[]) }.

sprinkler(S) :-
        cloudy(C),                  % C = f , t
        { S = sprinkler with p([f,t], [0.5,0.9,    % S = f
                                       0.5,0.1],   % S = t
                               [C])
        }.

rain(R) :-
        cloudy(C),            % C = f , t
        { R = rain with p([f,t], [0.8,0.2,   % R = f
                                  0.2,0.8],  % R = t
                          [C])
        }.

wet_grass(W) :-
        sprinkler(S),
        rain(R),
        { W = wet with p([f,t],
                 /* S/R = f/f, f/t, t/f, t/t */
                         [1.0, 0.1, 0.1, 0.01,  % W = f
                          0.0, 0.9, 0.9, 0.99], % W = t
                         [S,R])
        }.
```

[22] The implementation adds evidence declared in the input program to the store at compile time.

[23] taken from the examples in the CLP($\mathcal{BN}$) system

[24] We include comments for better readability, as CLP($\mathcal{BN}$) swaps rows and columns of CPTs compared to the notation in Figure 4.

In the clause for the top node cloudy, the body consists of a single constraint atom that constrains the logical variable `C` to the value of the random variable `cloudy`. This random variable takes values `f` or `t` with probability 0.5 each, and has an empty parent list. Note that within constraint atoms, the `=` sign does not denote Prolog unification, but an equality constraint between a logical variable and the value of a random variable. The clause for `sprinkler` calls `cloudy(C)`, thus setting up a constraint between `C` and the `cloudy` random variable, and then uses `C` as the only parent of the random variable `sprinkler` it defines. The first column of the CPT corresponds to the first parent value, the first row to the first child value, and so on, i.e., in case of `cloudy=f`, the probability of `sprinkler=f` is 0.5, whereas for `cloudy=t`, it is 0.9. The other two random variables are defined analogously, with their clauses again first calling the predicates for the parent variables to include the corresponding constraints. To answer the query `sprinkler(S)`, which asks for the marginal of the random variable `sprinkler`, CLP($\mathcal{BN}$) performs backward reasoning to find all constraints in the proof of the query, and thus the part of the Bayesian network relevant to compute the marginal. This first calls `cloudy(C)`, adding the constraint `C=cloudy` to the store (and thus the `cloudy` node to the BN), and then adds the constraint `S=sprinkler` to the store, and the `sprinkler` node with parent `cloudy` to the BN.

When defining relational models, random variables can be parameterized by logical variables as in the following clause from the school example included in the implementation:

```
registration_grade(R, Grade) :-
   registration(R, C, S),
   course_difficulty(C, Dif),
   student_intelligence(S, Int),
   grade_table(Int, Dif, Table),
   { Grade = grade(R) with Table }.

grade_table(I, D,
     p([a,b,c,d],
/* I,D =  h h   h m  h l  m h  m m  m l  l h  l m l l */
        [ 0.2,  0.7, 0.85, 0.1, 0.2, 0.5, 0.01, 0.05,0.1 , %a
          0.6, 0.25, 0.12, 0.3, 0.6,0.35,0.04, 0.15, 0.4 , %b
          0.15,0.04, 0.02, 0.4,0.15,0.12, 0.5, 0.6, 0.4,   %c
          0.05,0.01, 0.01, 0.2,0.05,0.03, 0.45, 0.2, 0.1 ],%d
        [I,D])).
```

Here, `registration/3` is a purely logical predicate linking a registration `R` to a course `C` and a student `S`. The predicates `course_difficulty` and `student_intelligence` define distributions over possible values `h(igh)`, `m(edium)`, and `l(ow)` for the difficulty `Dif` of course `C` and the intelligence `Int` of student `S`, respectively. For each grounding `r` of the variable `R` in the database of registrations, this clause defines a random variable `grade(r)` with values `a`, `b`, `c` and `d` that depends on the difficulty of the corresponding course and the intelligence of the corresponding student. In this case, the CPT itself is not defined within the constraint atom, but provided by a Prolog predicate binding it to a logical variable.

Defining aggregation using second order predicates is straightforward in CLP($\mathcal{BN}$), as random variables and constraints are part of the object level

vocabulary. For instance, the following clause defines the performance level of a student based on the average of his grades:

```
student_level(S,L) :-
  findall(G,(registration(R,_,S),registration_grade(R,G)),Grades),
  avg_grade(Grades,Avg),
  level_table(T),
  { L = level(S) with p([h,m,l],T,[Avg])}.
```

Here, `avg_grade/2` sets up a new random variable whose value is the suitably defined average of the grade list `Grades` (and which thus has a deterministic CPT) and constrains `Avg` to that random variable, and `level_table` provides the list of CPT entries specifying how the level depends on this average. We refer to Santos Costa et al. [2008] for a discussion of the inference challenges aggregates raise.

Despite the differences in syntax, probabilistic primitives, and inference between CLP($\mathcal{BN}$) and probabilistic extensions of Prolog following the distribution semantics, there are also many commonalities between those. As we discussed above, conditional probability tables can be represented using annotated disjunctions, and it is thus possible to transform CLP($\mathcal{BN}$) clauses into Prolog programs using annotated disjunctions. On the other hand, Santos Costa and Paes [2009] discuss the relation between PRISM and CLP($\mathcal{BN}$) based on a number of PRISM programs that they map into CLP($\mathcal{BN}$) programs.

6 Probabilistic Programming Concepts and Inference

We round off this survey by summarizing the relations between the dimensions of SUCC inference as discussed in Section 3 and the probabilistic programming concepts identified in Section 4. On the probabilistic side, we focus on *exact inference* versus *sampling*, as conclusions for exact inference carry over to approximate inference with bounds in most cases. On the logical side, we focus on *forward* versus *backward* reasoning, as conclusions for backward reasoning carry over to the approach using weighted model counting. We provide an overview in Table 2, where we omit the concepts *unknown objects*, as those are typically simulated via flexible probabilities and/or continuous distributions, and *constraints*, as those have not yet been considered during inference. For *generalized labels*, we focus on aProbLog, as it is closer to the distribution semantics than Dyna, due to its semantics based on worlds rather than derivations. We do not include MCMC here, as existing MCMC approaches in the context of the distribution semantics are limited to the basic case of definite clause programs without additional concepts.

Dimensions of inference: The main difference between exact inference and sampling is that the former has to consider *all* possible worlds or all proofs of the query, whereas the latter always considers one possible world or proof in isolation. As *second order predicates* and *time and dynamics* can increase the number of proofs exponentially (in the length of the answer list or the number of time steps), they are more easily handled by sampling based approaches, though

	Flexible Probabilities	Continuous Distributions	Stochastic Memoization	Negation as Failure	2nd Order Predicates	Meta-Calls	Time and Dynamics	Generalized Labels (aProbLog)
forward exact	no	no	with	yes	yes*	no	yes*	yes
backward exact	yes	limited	with or without	yes	yes*	yes	yes*	yes
forward sampling	no	yes	with	yes	yes	no	yes	n.a.
backward sampling	yes	yes	with or without	yes	yes	yes	yes	n.a.

Table 2: Relation between key probabilistic programming concepts and main dimensions of inference; see Section 6 for details. (* number of proofs/worlds exponential in length of answer list or time sequence)

tabling can significantly improve performance of exact inference in dynamic domains. Sampling based approaches do not directly apply for *generalized labels*, as sampling exploits the probabilistic semantics of fact labels.

The main difference between forward and backward reasoning is that the former generates all consequences of the probabilistic logic program, whereas the latter is query-driven and only considers relevant consequences, which can drastically improve efficiency. This difference is well-known in logic programming, and becomes even more important in the probabilistic setting, where we are interested in not just a single world or proof, but in all possible worlds or all proofs. The fact that backward reasoning is query-driven makes it well-suited for *flexible probabilities* and *meta-calls*, which cannot directly be handled in forward reasoning. The reason is that the corresponding subgoals have an infinite number of groundings, among which backward reasoning easily picks the relevant ones, which forward reasoning cannot do. The same effect makes it necessary to use *stochastic memoization* in forward reasoning, while backward reasoning can support dememoization (as in PRISM) as well as memoization (as in the various ICL, ProbLog and LPAD systems).

The roots of the distribution semantics in logic programming become apparent when considering inference for the two remaining key concepts, *negation as failure* and continuous distributions as provided by *distributional clauses*. While the logic concept of negation as failure is naturally supported in all combinations of exact inference or sampling and forward or backward reasoning, the probabilistic concept of continuous distributions is much more challenging, and only practical in sampling-based approaches.

Inference approaches: More specifically, *exact inference using forward reasoning* in the form discussed in Section 3.1 can be used for all programs with finitely many finite worlds, which (a) excludes the use of non-ground facts without explicitly given domains, flexible probabilities, meta-calls and continuous probabilities, and (b) requires stochastic memoization. As this approach additionally suffers from having to enumerate all possible worlds, it is not used in practice.[25]

Exact inference using backward reasoning is the most widely supported inference technique in probabilistic logic programming, provided by AILog2, PRISM, ProbLog1, cplint, PITA and MetaProbLog. PRISM never uses stochastic memoization, whereas the other systems always use it. Only very limited forms of continuous distributions can be supported, cf. the work on Hybrid ProbLog [Gutmann et al., 2010]. All other concepts can be supported, but implementations differ in the ones they cover. Negation as failure is supported in all implementations. In addition, AILog2 and cplint support flexible probabilities, MetaProbLog supports flexible probabilities and meta-calls, and ProbLog1 supports flexible probabilities, limited use of continuous distributions (Hybrid ProbLog) and generalized labels (aProbLog). Approximate inference with bounds using backward reasoning is available in ProbLog1 and cplint, but restricted to definite clause programs, as the use of negation as failure complicates proof finding (as discussed in Section 4.6). As the WMC approach as implemented in ProbLog2 uses backward inference to determine the relevant grounding, that is, the groundings of clauses that appear in some proof of a query, the same observations as for exact backward inference apply in this case as well. ProbLog2 supports flexible probabilities and negation as failure.

Forward sampling in its simplest form as discussed in Section 3.1 can be used with programs whose worlds are all finite, which excludes the use of non-ground facts without explicitly given domains, flexible probabilities, and meta-calls, and requires stochastic memoization. In contrast to exact forward inference, forward sampling does support continuous distributions, as only one value is considered at a time. None of the probabilistic logic programming systems discussed here implement forward sampling.

Backward sampling is the most flexible approach and can in principle deal with all concepts except generalized labels. Backward sampling approaches are provided by ProbLog1 and cplint, which both support flexible probabilities and negation as failure. PRISM has a builtin for sampling the outcome of a query using backward reasoning, but does not use it for probability estimation.

7 Conclusions

Probabilistic programming is a rapidly developing field of research as witnessed by the many probabilistic programming languages and primitives that have been introduced over the past few years. In this paper, we have attempted to provide a gentle introduction to this field by focussing on probabilistic *logic* programming languages and identifying the underlying probabilistic concepts that these languages support. The same concept (e.g., probabilistic choice) can be realized using different syntactic primitives (e.g., switches, annotated disjunctions,

[25] Dyna's exact inference is based on forward reasoning, but uses a different type of algorithm that propagates value updates using forward reasoning based on an agenda of pending updates.

etc.) leading to differences in the probabilistic programming languages. Probabilistic programming implementations not only differ in the primitives they provide but also in the way they perform probabilistic inference. Inference is a central concern in these languages, as probabilistic inference is computationally expensive. We have therefore also presented various probabilistic inference mechanisms and discussed their suitability for supporting the probabilistic programming concepts. This in turn allowed us to position different languages and implementations, leading to a broad survey of the state-of-the-art in probabilistic logic programming.

Acknowledgements

The authors are indebted to Bernd Gutmann and Ingo Thon for participating in many discussions, and contributing several ideas during the early stages of the research that finally led to this paper. Angelika Kimmig is supported by the Flemish Research Foundation (FWO-Vlaanderen).

References

Nicos Angelopoulos and James Cussens. On the implementation of MCMC proposals over stochastic logic programs. In *Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS-04)*, 2004.

Nimar S. Arora, Rodrigo de Salvo Braz, Erik B. Sudderth, and Stuart J. Russell. Gibbs sampling in open-universe stochastic languages. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI-10)*, 2010.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD symposium on Principles of Database Systems (PODS-86)*, 1986.

Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming (TPLP)*, 9(1): 57–144, 2009.

Stefano Bragaglia and Fabrizio Riguzzi. Approximate inference for logic programs with annotated disjunctions. In *Revised papers of the 20th International Conference on Inductive Logic Programming (ILP-10)*, 2011.

Matthias Broecheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI-10)*, 2010.

Shay B. Cohen, Robert J. Simmons, and Noah A. Smith. Dynamic programming algorithms as products of weighted logic programs. In *Proceedings of the 24th International Conference on Logic Programming (ICLP-08)*, 2008.

J. Cussens. Integrating by separating: Combining probability and logic with ICL, PRISM and SLPs. APRIL II project report, 2005.

N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB-04)*, 2004.

Evgeny Dantsin. Probabilistic logic programs and their semantics. In *Proceedings of the First Russian Conference on Logic Programming*, 1991.

L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.

L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008.

Luc De Raedt and Kristian Kersting. Probabilistic logic learning. *SIGKDD Explorations*, 5(1):31–48, 2003.

J. Eisner, E. Goldlust, and N. Smith. 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)*, 2005.

Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2011.

Daan Fierens, Hendrik Blockeel, Maurice Bruynooghe, and Jan Ramon. Logical Bayesian networks and their relation to other probabilistic logical models. In *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP-05)*, 2005.

Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI-11)*, 2011.

Daan Fierens, Guy Van den Broeck, Maurice Bruynooghe, and Luc De Raedt. Constraints for probabilistic logic programming. In *Proceedings of the NIPS Probabilistic Programming Workshop*, 2012.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming (TPLP)*, 2014. (Accepted).

Peter A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

N. Fuhr. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *JASIS*, 51(2):95–110, 2000.

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

L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In Getoor and Taskar [2007], pages 129–174.

N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence (UAI-08)*, 2008.

Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending ProbLog with continuous distributions. In *Proccedings of the 20th International Conference on Inductive Logic Programming (ILP-10)*, 2010.

Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming (TPLP)*, 11((4–5)):663–680, July 2011.

P. Haddawy. Generating Bayesian networks from probabilistic logic knowledge bases. In *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, 1994.

Arjen Hommersom and Peter J. F. Lucas. Generalising the interaction rules in probabilistic logic. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, 2011.

Arjen Hommersom, Nivea de Carvalho Ferreira, and Peter J. F. Lucas. Integrating logical reasoning and probabilistic chain graphs. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD-09)*, 2009.

Manfred Jaeger. Relational Bayesian networks. In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)*, 1997.

Manfred Jaeger. Model-theoretic expressivity analysis. In De Raedt et al. [2008], pages 325–339.

Kristian Kersting. Lifted probabilistic inference. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI-12)*, 2012.

Kristian Kersting and Luc De Raedt. Basic principles of learning Bayesian logic programs. In De Raedt et al. [2008], pages 189–221.

Kristian Kersting, Luc De Raedt, and Tapani Raiko. Logical hidden Markov models. *J. Artif. Intell. Res. (JAIR)*, 25:425–456, 2006.

A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In *Proceedings of the 24th International Conference on Logic Programming (ICLP-08)*, 2008.

Angelika Kimmig, Bernd Gutmann, and Vìtor Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *Proceedings of the International Workshop on Statistical Relational Learning (SRL-2009)*, 2009.

Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming (TPLP)*, 11: 235–262, 2011a.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, 2011b.

J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2. edition, 1989.

Theofrastos Mantadelis and Gerda Janssens. Nesting probabilistic inference. In *Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS-11)*, 2011.

B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.

Bogdan Moldovan, Ingo Thon, Jesse Davis, and Luc De Raedt. MCMC estimation of conditional probabilities in probabilistic programming languages. In *Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-13)*, 2013.

S. H Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

Sriraam Natarajan, Prasad Tadepalli, Eric Altendorf, Thomas G. Dietterich, Alan Fern, and Angelo C. Restificar. Learning first-order probabilistic models with combining rules. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-05)*, 2005.

Davide Nitti, Tinne De Laet, and Luc De Raedt. A particle filter for hybrid relational domains. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-13)*, 2013.

A. Pfeffer. IBAL: A probabilistic rational programming language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

D. Poole. Logic programming, abduction and probability. In *Fifth Generation Computing Systems*, pages 530–538, 1992.

D. Poole. The independent choice logic and beyond. In De Raedt et al. [2008], pages 222–243.

David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

David Poole. Abducing through negation as failure: stable models within the independent choice logic. *Journal of Logic Programming*, 44(1-3):5–35, 2000.

David Poole. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.

David Poole. Probabilistic programming languages: Independent choices and deterministic systems. In H. Geffner R. Dechter and J.Y. Halpern, editors, *Heuristics, Probability and Causality: A Tribute to Judea Pearl*, pages 253–269. College Publications, 2010.

Hoifung Poon and Pedro Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, 2006.

Joris Renkens, Guy Van den Broeck, and Siegfried Nijssen. k-optimal: A novel approximate inference algorithm for ProbLog. *Machine Learning*, 89(3):215–231, July 2012.

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

Fabrizio Riguzzi. Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009.

Fabrizio Riguzzi. *cplint Manual*, 2013a. `https://sites.google.com/a/unife.it/ml/cplint/cplint-manual`.

Fabrizio Riguzzi. Mcintyre: A monte carlo system for probabilistic logic programming. *Fundam. Inform.*, 124(4):521–541, 2013b.

Fabrizio Riguzzi. Speeding up inference for probabilistic logic programs. *The Computer Journal*, 2013c. (Online first).

Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming (TPLP)*, 11(4-5):433–449, 2011.

D. Roy, V. Mansinghka, J. Winn, D. McAllester, and D. Tenenbaum, editors. *Probabilistic Programming*, 2008. NIPS Workshop.

S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

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

Vìtor Santos Costa and Aline Paes. On the relationship between PRISM and CLP(*BN*). In *Proceedings of the International Workshop on Statistical Relational Learning (SRL-2009)*, 2009.

T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP-95)*, 1995.

T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res. (JAIR)*, 15:391–454, 2001.

Taisuke Sato. A general MCMC method for bayesian inference in logic-based probabilistic modeling. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, 2011.

Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997.

Taisuke Sato, Yoshitaka Kameya, and Neng-Fa Zhou. Generative modeling with failure in prism. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 847–852. Professional Book Center, 2005. ISBN 0938075934.

Ingo Thon, Niels Landwehr, and Luc De Raedt. Stochastic relational processes: Efficient inference and applications. *Machine Learning*, 82(2):239–272, 2011.

Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

Guy Van den Broeck, Ingo Thon, Martijn van Otterlo, and Luc De Raedt. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, 2010.

Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, 2011.

J. Vennekens. *Algebraic and logical study of constructive processes in knowledge representation*. PhD thesis, K.U. Leuven, 2007.

Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *Proceedings of the 20th International Conference on Logic Programming (ICLP-04)*, 2004.

Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming (TPLP)*, 9(3):245–308, 2009.

William Yang Wang, Kathryn Mazaitis, and William W. Cohen. Programming with personalized pagerank: a locally groundable first-order probabilistic logic. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM-13)*, 2013.

A Logic Programming Basics

The basic building blocks of logic programs are *variables* (denoted by strings starting with upper case letters), *constants*, *functors* and *predicates* (all denoted by strings starting with lower case letters). A *term* is a variable, a constant, or a functor $\mathtt{f}$ of *arity* n followed by n terms $\mathtt{t_i}$, i.e., $\mathtt{f}(\mathtt{t_1}, ..., \mathtt{t_n})$. An *atom* is a predicate p of arity n followed by n terms $\mathtt{t_i}$, i.e., $\mathtt{p}(\mathtt{t_1}, ..., \mathtt{t_n})$. A predicate p of arity n is also written as p/n. A *literal* is an atom or a negated atom $\mathtt{not}(\mathtt{p}(\mathtt{t_1}, ..., \mathtt{t_n}))$. A *definite clause* is a universally quantified expression of the form $h :- b_1, ..., b_n$ where h and the b_i are atoms. h is called the *head* of the clause, and $b_1, ..., b_n$ its *body*. Informally, the meaning of such a clause is that if all the b_i are true, h has to be true as well. A *normal clause* is a universally

quantified expression of the form $h :- b_1, ..., b_n$ where h is an atom and the b_i are literals. If $n = 0$, a clause is called *fact* and simply written as h. A *definite clause program* or *logic program* for short is a finite set of definite clauses. A *normal logic program* is a finite set of normal clauses. A *substitution* θ is an expression of the form $\{V_1/t_1, ..., V_m/t_m\}$ where the V_i are different variables and the t_i are terms. Applying a substitution θ to an expression e (term or clause) yields the *instantiated* expression $e\theta$ where all variables V_i in e have been simultaneously replaced by their corresponding terms t_i in θ. If an expression does not contain variables it is *ground*. Two expressions e_1 and e_2 can be *unified* if and only if there are substitutions θ_1 and θ_2 such that $e_1\theta_1 = e_2\theta_2$. In Prolog, unification is written using = as an infix predicate.

The *Herbrand base* of a logic program is the set of ground atoms that can be constructed using the predicates, functors and constants occurring in the program[26]. Subsets of the Herbrand base are called *Herbrand interpretations*. A Herbrand interpretation is a *model* of a clause $\mathtt{h} \ :- \ \mathtt{b_1}, \ldots, \mathtt{b_n}.$ if for every substitution θ such that all $b_i\theta$ are in the interpretation, $h\theta$ is in the interpretation as well. It is a model of a logic program if it is a model of all clauses in the program. The model-theoretic semantics of a definite clause program is given by its smallest Herbrand model with respect to set inclusion, the so-called *least Herbrand model* (which is unique). We say that a logic program P *entails* an atom a, denoted $P \models a$, if and only if a is true in the least Herbrand model of P.

The main inference task of a logic programming system is to determine whether a given atom, also called *query* (or *goal*), is true in the least Herbrand model of a logic program. If the answer is yes (or no), we also say that the query *succeeds* (or *fails*). If such a query is not ground, inference asks for the existence of an *answer substitution*, that is, a substitution that grounds the query into an atom that is part of the least Herbrand model.

Normal logic programs use the notion of *negation as failure*, that is, for a ground atom a, $not(a)$ is true exactly if a cannot be proven in the program. They are not guaranteed to have a unique minimal Herbrand model. Various ways to define the canonical model of such programs have been studied; see, e.g., [Lloyd, 1989, Chapter 3] for an overview.

B Annotated Disjunctions and Probabilistic Facts

As mentioned in Section 2.2, each annotated disjunction can be equivalently represented using a set of probabilistic facts and deterministic clauses. Using probabilistic facts is not sufficient, as those correspond to independent random variables. For instance, using probabilistic facts

$$\tfrac{1}{3} :: \mathtt{color(green)}. \qquad \tfrac{1}{3} :: \mathtt{color(red)}. \qquad \tfrac{1}{3} :: \mathtt{color(blue)}.$$

the probability of `color(green)`, `color(red)` and `color(blue)` all being true is 1/27, whereas it is 0 for the annotated disjunction $\frac{1}{3}$ :: `color(green)`; $\frac{1}{3}$ :: `color(red)`; $\frac{1}{3}$ :: `color(blue)`. On the other hand, we can exploit the fact that negation of probabilistic facts is easily handled under the distribution seman-

[26] If the program does not contain constants, one arbitrary constant is added.

tics[27] to encode an AD by simulating a sequential choice mechanism[28]. With this encoding, the three possible outcomes are mutually exclusive as in the AD and exactly one will be true in any possible world:

```
1/3 :: sw_1(color(green)).   1/2 :: sw_1(color(red)).   1 :: sw_1(color(blue)).

color(green) :- sw_1(color(green)).
color(red)   :- not(sw_1(color(green))), sw_1(color(red)).
color(blue)  :- not(sw_1(color(green))), not(sw_1(color(red))), sw_1(color(blue)).
```

Note that the probabilities have been adapted to reproduce the probabilities of the different head atoms; we discuss the details of this adaptation below.[29] This mapping follows the general idea of representing a probabilistic model in an *augmented space* where random variables can be assumed independent, while capturing the dependencies in the deterministic part of the program [Poole, 2010].

For non-ground ADs, all logical variables have to be included in the probabilistic facts to ensure that all groundings correspond to independent random events. For instance, the AD $(\frac{1}{2}$ :: `color(green)`; $\frac{1}{2}$ :: `color(red)`$)$:− `ball(Ball)` would be represented as

```
1/2 :: sw_1(color(green), Ball).   1 :: sw_1(color(red), Ball).

color(green) :- ball(Ball), sw_1(color(green), Ball).
color(red)   :- ball(Ball), not(sw_1(color(green), Ball)), sw_1(color(red), Ball).
```

As this example suggests, annotated disjunctions can be expressed using probabilistic facts by representing each annotated disjunction using the set of probabilistic facts $\tilde{p}_i :: sw\_id(h_i, v_1, \ldots, v_f)$ and the following clauses

$$h_i :- b_1, \cdots, b_m, not(sw\_id(h_1, v_1, \ldots, v_f)), ..., not(sw\_id(h_{i-1}, v_1, \ldots, v_f)), \\ sw\_id(h_i, v_1, \ldots, v_f) \tag{8}$$

where id is a unique identifier for a particular AD and $v_1, \ldots, v_f$ are the free variables in the body of the AD. The probability $\tilde{p}_1$ is defined as p_1 and for $i > 1$ it is

$$\tilde{p}_i := \begin{cases} p_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right)^{-1} & \text{if } p_i > 0 \\ 0 & \text{if } p_i = 0 \end{cases} . \tag{9}$$

One can recover the original probabilities from $\tilde{p}$ by setting $p_1 := \tilde{p}_1$ and iteratively applying the following transformation for $i = 2, 3, \ldots, n$

$$p_i := \tilde{p}_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right) . \tag{10}$$

Equation (9) and (10) together define a bijection between p and $\tilde{p}$ which allows one to use parameter learning in either representation and map learned probabilities onto the other representation. If the p_i sum to 1, it is possible to drop the last probabilistic fact $sw\_id(h_n)$ since its probability $\tilde{p}_n$ is 1.

[27] For a probabilistic fact `p::f`, `not(f)` succeeds in a possible world exactly if `f` is not among the probabilistic facts included in that world; cf. Section 4.6 for a more general discussion of negation.

[28] used, e.g., by Sato and Kameya [1997] with parameters learned from data

[29] This transformation is correct for computing success probabilities, but care has to be taken to accomodate for the additional random variables in MPE inference.