1 A Survey of Probabilistic Logic Programming

Fabrizio Riguzzi, Dipartimento di Matematica e Informatica, University of Ferrara, Ferrara, Italy
Theresa Swift, Coherent Knowledge Systems, Mercer Island, Washington, USA and NOVA LINCS Universidade Nova de Lisboa, Portugal

The combination of logic programming and probability has proven useful for modeling domains with complex and uncertain relationships among elements. Many probabilistic logic programming (PLP) semantics have been proposed, among these the distribution semantics has recently gained an increased attention and is adopted by many languages such as the Independent Choice Logic, PRISM, Logic Programs with Annotated Disjunctions, ProbLog and P-log.

This chapter reviews the distribution semantics, beginning in the simplest case with stratified Datalog programs, and showing how the definition is extended to programs that include function symbols and non-stratified negation. The languages that adopt the distribution semantics are also discussed and compared both to one another and to Bayesian networks. We then survey existing approaches for inference in PLP languages that follow the distribution semantics. We concentrate on the PRISM, ProbLog and PITA systems. The PRISM system was one of the first and can be applied when certain restrictions on the program hold. ProbLog introduced the use of Binary Decision Diagrams that provide a computational basis for removing these restrictions and so performing inference over more general classes of logic programs. PITA speeds up inference by using tabling and answer subsumption. It supports general probabilistic programs, but can easily be optimized for simpler settings and even possibilistic uncertain reasoning. The chapter also discusses the computational complexity of the various approaches together with techniques for limiting it by resorting to approximation.

1.1 Introduction

If inference is a central aspect of mathematical logic, then *uncertain inference* (a term coined by Henry Kyburg) is central to much of computational logic and to logic programming in particular. The term uncertain inference captures non-monotonic reasoning, fuzzy and possibilistic logic, as well as combinations of logic and probability. Intermixing probabilistic with logical inference is of particular interest for artificial intelligence tasks such as modeling agent behavior, diagnosing complex systems, assessing risk, and conducting structure learn-

ing. It is also useful for exploiting learned knowledge that it is itself probabilistic, such as used in medicine, bioinformatics, natural language parsing, marketing, and much else. These aspects have led to a huge amount of research into probabilistic formalisms such as Bayesian networks, Markov networks, and statistical learning techniques.

These trends have been reflected within the field of logic programming by various approaches to Probabilistic Logic Programming (PLP). These approaches include the languages and frameworks Probabilistic Logic Programs [Dantsin 1991], Probabilistic Horn Abduction [Poole 1993b], PRISM [Sato 1995], Independent Choice Logic [Poole 1997], pD [Fuhr 2000], Bayesian Logic Programs [Kersting and Raedt 2001], CLP(BN) [Costa et al. 2003], Logic Programs with Annotated Disjunctions [Vennekens et al. 2004], P-log [Baral et al. 2009], ProbLog [De Raedt et al. 2007] and CP-logic [Vennekens et al. 2009]. While such a profusion of approaches indicates a ferment of interest in PLP, the question arises that if there are so many different languages, is any of them the right one to use? And why should PLP be used at all as opposed to Bayesian networks or other more popular approaches?

Fortunately, most of these approaches have similarities that can be brought into focus using various forms of the distribution semantics [Sato 1995][1]. Under the distribution semantics, a logic program defines a probability distribution over a set, each element of which is a normal logic program (termed a *world*). When there are a finite number of such worlds, as is the case with Datalog programs, the probability of an atom A is directly based on the proportion of the worlds whose model contains A as true. It can immediately be seen that distribution semantics are types of frequency semantics (cf. e.g., [Halpern 2003]). By replacing worlds with sets of worlds, the same idea can be used to construct probabilities for atoms in programs that have function symbols and so may have an infinite number of worlds. Given these similarities, various forms of the distribution semantics differ on how a model is associated with a world: a model may be a (minimal) stratified model, a stable model, or even a well-founded model. Finally, the semantics of Bayesian networks can be shown to be equivalent to a restricted class of probabilistic logic program under the distribution semantics, indicating that approaches to PLP are at least as powerful as Bayesian networks.

For these reasons, this chapter uses the distribution semantics as an organizing principle to present an introduction to PLP. Our focus on the distribution semantics and on the problem of inference distinguishes this survey from [De Raedt and Kimmig 2015], a useful overview that focuses more on the actual programming in PLP. Accordingly, Section 1.2 starts with examples of some PLP languages. Section 1.3 then formally presents the distribution semantics in stages. The simplest case — that of Datalog programs with a single stratified model — is presented first in Section 1.3.1; using this basis the languages of Section 1.2 are shown to be expressively equivalent for stratified Datalog. Next, Section 1.3.3 extends the distribution semantics for programs with function symbols, by associating each *explanation* of a query (a set

[1] In this chapter the term *distribution semantics* is used in different contexts to refer both to a particular semantics and to a family of related semantics.

of probabilistic facts needed to prove the query) with a set of worlds, and constructing probability distributions on these sets. As a final extension, the assumption of a single stratified model for each world is lifted (Section 1.3.4).

Section 1.4 discusses semantics for probabilistic logics that are alternative to the distribution semantics. Section 1.5 then discusses PLP languages that are closely related to Bayesian networks and shows how Bayesian networks are equivalent to special cases of PLPs.

The distribution semantics is essentially model-theoretic; Section 1.6 discusses how inferences can be made from probabilistic logic programs. First, the relevant complexity results are recalled in Section 1.6.1: given that probabilistic logic programs are as expressive as Bayesian networks, query answering in probabilistic logic programs is easily seen to be NP-hard, and in fact is #*P*-complete. Current exact inferencing techniques for general probabistic logic programs, such as the use of Binary Decision Diagrams as pioneered in the ProbLog system [Kimmig et al. 2011] are discussed in Section 1.6.2. Section 1.6.3 discusses special cases of probabilistic logic programs for which inferencing is tractable and that have been exploited especially by the PRISM system [Sato et al. 2010]. Section 1.7 concludes the chapter with a final discussion.

1.1.1 Background and Assumptions

This survey can be read either by those with familiarity with logic programming who want to learn about its probabilistic extensions, or by those with background in probabilistic programming systems who want to learn how probabilistic reasoning can be modeled and implemented in logic programming. In terms of logic programming, we assume a basic familiarity with the syntax and terminology of Prolog/ASP, along with the general notion that programs may have different types of models depending on how negation is used. For instance, there are two main semantics for negation that are implemented in current logic programming systems: stable models and well-founded models, see [Truszczynski 2018] for a gentle introduction. These two semantics coincide for programs that either do not use negation (definite programs), or where negation is well-behaved (stratified programs) [Truszczynski 2018]. Technical details about these models, such as how they are constructed via a fixed point, are not required. In terms of probability theory, we usually assume only a basic understanding of discrete distributions; definitions of other concepts are recalled when needed.

1.2 Languages with the Distribution Semantics

The languages following distribution semantics largely differ in how they encode choices for clauses, and how the probabilities for these choices are stated. In all languages, however, choices are independent from each other. As will be shown in Section 1.3.2, as long as models for the various types of programs are associated to worlds in the same manner – they all have the same expressive power. This fact shows that the differences in the languages are syntactic, and also justifies speaking of *the* distribution semantics.

1.2.0.1 Probabilistic Horn Abduction

In Probabilistic Horn Abduction (PHA) [Poole 1993b] and Independent Choice Logic (ICL) [Poole 1997], alternatives are expressed by facts, called *disjoint-statements*, having the form

$$disjoint([A_1 : p_1, \ldots, A_n : p_n]).$$

where each A_i is a logical atom and each p_i a number in $[0,1]$ such that $\sum_{i=1}^{n} p_i = 1$. Such a statement can be interpreted in terms of its ground instantiations: for each substitution θ grounding the atoms of the statement, the $A_i\theta$s are random alternatives and $A_i\theta$ is true with probability p_i. Each world is obtained by selecting one atom from each grounding of each disjoint-statement in the program. In practice, each ground instantiation of a disjoint statement corresponds to a random variable with as many values as the alternatives in the statement. The variables corresponding to different instantiations of the same disjoint statement are *independent and identically distributed (iid)*.

Example 1. *The following PHA/ICL program encodes the fact that a person sneezes if he has the flu and this is the active cause of sneezing, or if he has hay fever and hay fever is the active cause for sneezing:*

$$\begin{array}{l} sneezing(X) \text{ :- } flu(X), flu\_sneezing(X). \\ sneezing(X) \text{ :- } hay\_fever(X), hay\_fever\_sneezing(X). \\ flu(bob). \\ hay\_fever(bob). \end{array}$$

$$\begin{array}{ll} disjoint([flu\_sneezing(X) : 0.7, null : 0.3]). & (C_1) \\ disjoint([hay\_fever\_sneezing(X) : 0.8, null : 0.2]). & (C_2) \end{array}$$

Here, and for the other languages based on the distribution semantics, the atom null does not appear in the body of any clause and is used to represent an alternative in which no atom is selected.

1.2.0.2 PRISM

The language PRISM [Sato and Kameya 1997] is similar to PHA/ICL but introduces random facts via the predicate $msw/3$ (multi-switch):

$$msw(SwitchName, TrialId, Value).$$

The first argument of this predicate is a *random switch name*, a term representing a set of discrete random variables; the second argument is an integer, the *trial id*; and the third argument represents a value for that variable. The set of possible values for a switch is defined by a fact of the form

$$values(SwitchName, [v_1, \ldots, v_n]).$$

where *SwitchName* is again a term representing a switch and each v_i is a term. Each ground pair $(SwitchName, TrialId)$ represents a distinct random variable and the set of random variables associated with the same switch are iid.

The probability distribution over the values of the random variables associated to *SwitchName* is defined by a directive of the form

$$:\text{- } set\_sw(SwitchName, [p_1, \ldots, p_n]).$$

where p_i is the probability that variable *SwitchName* takes value v_i. Each world is obtained by selecting one value for each trial id of each random switch.

Example 2. *The modeling of coin tosses shows differences in how the various PLP languages represent iid random variables. Suppose that coin c_1 is known not to be fair, but that all tosses of c_1 have the same probabilities of outcomes – in other words each toss of c_1 is taken from a family of iid random variables. This can be represented in PRISM as*

$$values(c_1, [head, tail]).$$
$$:\text{- } set\_sw(c_1, [0.4, 0.6])$$

Different tosses of c_1 can then be identified using the trial id *argument of* msw/3.

In PHA/ICL and many other PLP languages, each ground instantiation of a disjoint/1 *statement represents a distinct random variable, so that iid random variables need to be represented through the statement's instantiation patterns: e.g.,*

$$disjoint([coin(c_1, TossNumber, head) : 0.4, coin(c_1, TossNumber, tail) : 0.6]).$$

In practice, the PRISM systems accepts an $msw/2$ predicate whose atoms do not contain the trial id and for which each occurrence in a program is considered as being associated to a different new variable.

Example 3. *Example 1 can be encoded in PRISM as:*

$$sneezing(X) :\text{- } flu(X), msw(flu\_sneezing(X), 1).$$
$$sneezing(X) :\text{- } hay\_fever(X), msw(hay\_fever\_sneezing(X), 1).$$
$$flu(bob).$$
$$hay\_fever(bob).$$

$$values(flu\_sneezing(\_X), [1, 0]).$$
$$values(hay\_fever\_sneezing(\_X), [1, 0]).$$
$$:\text{- } set\_sw(flu\_sneezing(\_X), [0.7, 0.3]).$$
$$:\text{- } set\_sw(hay\_fever\_sneezing(\_X), [0.8, 0.2]).$$

1.2.0.3 Logic Programs with Annotated Disjunctions

In Logic Programs with Annotated Disjunctions (LPADs) [Vennekens et al. 2004], the alternatives are expressed by means of annotated disjunctive heads of clauses. An *annotated disjunctive clause* has the form

$$H_{i1} : p_{i1}; \ldots; H_{in_i} : p_{in_i} \text{ :- } B_{i1}, \ldots, B_{im_i}$$

where $H_{i1}, \ldots, H_{in_i}$ are logical atoms, $B_{i1}, \ldots, B_{im_i}$ are logical literals and $p_{i1}, \ldots, p_{in_i}$ are real numbers in the interval $[0,1]$ such that $\sum_{k=1}^{n_i} p_{ik} = 1$. Each world is obtained by selecting one atom from the head of each grounding of each annotated disjunctive clause[2].

Example 4. *Example 1 can be expressed in LPADs as:*

$$\begin{array}{ll} \mathit{sneezing}(X) : 0.7 \vee \mathit{null} : 0.3 \text{ :- } \mathit{flu}(X). & (C_1) \\ \mathit{sneezing}(X) : 0.8 \vee \mathit{null} : 0.2 \text{ :- } \mathit{hay\_fever}(X). & (C_2) \\ \mathit{flu}(\mathit{bob}). & \\ \mathit{hay\_fever}(\mathit{bob}). & \end{array}$$

1.2.0.4 ProbLog

The design of ProbLog [De Raedt et al. 2007] was motivated by the desire to make as simple a probabilistic extension of Prolog as possible. In ProbLog alternatives are expressed by *probabilistic facts* of the form

$$p_i :: A_i$$

where $p_i \in [0,1]$ and A_i is an atom, meaning that each ground instantiation $A_i\theta$ of A_i is true with probability p_i and false with probability $1 - p_i$. Each world is obtained by selecting or rejecting each grounding of all probabilistic facts.

Example 5. *Example 1 can be expressed in ProbLog as:*

$$\begin{array}{l} \mathit{sneezing}(X) \text{ :- } \mathit{flu}(X), \mathit{flu\_sneezing}(X). \\ \mathit{sneezing}(X) \text{ :- } \mathit{hay\_fever}(X), \mathit{hay\_fever\_sneezing}(X). \\ \mathit{flu}(\mathit{bob}). \\ \mathit{hay\_fever}(\mathit{bob}). \\ 0.7 :: \mathit{flu\_sneezing}(X). \\ 0.8 :: \mathit{hay\_fever\_sneezing}(X). \end{array}$$

As for ICL, in LPADs and ProbLog each grounding of a probabilistic clause is associated to a random variable with as many values as head disjuncts for LPADs and with two values for ProbLog. The random variables corresponding to different instantiations of a probabilistic clause are iid.

[2] CP-logic [Vennekens et al. 2009] has a similar syntax to LPADs, and the semantics of both languages coincide for stratified Datalog programs.

1.3 Defining the Distribution Semantics

In presenting the distribution semantics, we use the term *probabilistic construct* to refer to disjoint-statements, multi-switches, annotated disjunctive clauses, and probabilistic facts, in order to discuss their common properties.

The distribution semantics applies to unrestricted normal logic programs. Nonetheless, for the purposes of explanation, we begin in Section 1.3.1 by making two simplifications.

- *Datalog Programs*: if a program has no function symbols, the Herbrand universe is finite and so is the set of groundings of each probabilistic construct.
- *Stratified Programs*: these programs have either a total well founded model [Van Gelder et al. 1991] or equivalently a single stable model [Gelfond and Lifschitz 1988][3].

With the distribution semantics thus defined, Section 1.3.2 discusses the relationships among the languages presented in Section 1.2. Afterwards, the restriction to Datalog programs is lifted in Section 1.3.3, while the restriction of stratification is lifted in Section 1.3.4. We note that throughout this section, all probabilities distributions are discrete; however continuous probability distributions have also been used with the distribution semantics [Gutmann et al. 2011a, Islam et al. 2012].

1.3.1 The Distribution Semantics for Stratified Datalog Programs

An *atomic choice* is the selection of the i-th atom for grounding $C\theta$ of a probabilistic construct C. It is denoted with the triple (C,θ,i) where C is a clause, θ is a grounding substitution and i is the index of the alternative atom chosen. In Example 1, $(C_1,\{X/bob\},1)$ is an atomic choice relative to disjoint-statement

$$C_1 = disjoint([flu\_sneezing(X) : 0.7, null : 0.3]).$$

denoting the selection of atom $flu\_sneezing(bob)$. Atomic choices for other languages are made similarly: for instance, an atomic choice for a ProbLog fact $p :: foo(X)$ and a substitution $\theta = \{X/a\}$ can be obtained by interpreting the fact as $C = foo(X) : p \vee null : 1-p$, so $(C,\theta,1)$ selects atom $A\theta$ while $(C,\theta,2)$ selects atom $null$.

A set of atomic choices is *consistent* if it does not contain two atomic choices (C,θ,i) and (C,θ,j) with $i \neq j$ (only one alternative is selected for a ground probabilistic construct).

A *composite choice* κ is a consistent set of atomic choices, i.e., if $(C,\theta,i) \in \kappa$ and $(C,\theta,j) \in \kappa$ then $i = j$. In Example 1, the set of atomic choices $\kappa = \{(C_1,\{X/bob\},1),(C_1,\{X/bob\},2)\}$

[3] This restriction is sometimes called *soundness* in the PLP literature. There have been various definitions of stratification in the literature, which involve various types of negative self-dependency occurring in the derivation of an atom A. Among the most general is that of [Przymusinski 1989], in which a program is dynamically stratified iff it has a two-valued well-founded model. This notion of stratification is used (sometimes implicitly) in this chapter.

is not consistent. The probability of composite choice κ is

$$P(\kappa) = \prod_{(C,\theta,i)\in\kappa} p_i$$

where p_i is the probability of the i-th alternative for probabilistic construct C.

A *selection* σ is a total composite choice, i.e., it contains one atomic choice for every grounding of each probabilistic construct. A selection in Example 1 is

$$\sigma_1 = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\}.$$

A *world* w_σ is a normal logic program that is identified by a selection σ. The world w_σ is formed by replacing probabilistic constructs from the program with the non probabilistic constructs corresponding to each atomic choice of σ. In other words, for each atomic choice (C,θ,i), a ground clause or fact is obtained from $C\theta$ by selecting the i-th alternative from the construct. For instance, given the previous selection σ_1, the atoms *flu_sneezing(bob)* and *hay_fever_sneezing(bob)* would be added to the first four clauses of Example 1 to make w_{σ_1}. Note that a world is a normal logic program, which may include rules and facts, see Examples 6 and 7 below. For the LPAD of Example 4, the selection $\sigma_1 = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\}$ yields the clauses

$$sneezing(bob) \text{ :- } flu(bob).$$
$$sneezing(bob) \text{ :- } hay\_fever(bob).$$

that are included in w_{σ_1}.

The probability of a world w_σ is

$$P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} p_i.$$

Since in this section we are assuming Datalog programs, the set of groundings of each probabilistic construct is finite, and so is the set of worlds W_T. Accordingly, for a probabilistic logic program T, $W_T = \{w_1, \ldots, w_m\}$. Moreover, $P(w)$ is a distribution over worlds: $\sum_{w\in W_T} P(w) = 1$.

Let Q be a query in the form of a ground atom. We define the conditional probability of Q given a world w as: $P(Q|w) = 1$ if Q is true in the model of w and 0 otherwise. Since in this section we consider only stratified negation (sound programs), w has only one two-valued model and Q can be only true or false in it. The probability of Q can thus be computed by summing out the worlds from the joint distribution of the query and the worlds:

$$P(Q) = \sum_w P(Q,w) = \sum_w P(Q|w)P(w) = \sum_{w\models Q} P(w)$$

Example 6. *The PHA/ICL program of Example 1 has four worlds* $\{w_1, w_2, w_3, w_4\}$*, each containing the certain (non-probabilistic) clauses:*

$$\begin{array}{l} sneezing(X) \text{ :- } flu(X), flu\_sneezing(X). \\ sneezing(X) \text{ :- } hay\_fever(X), hay\_fever\_sneezing(X). \\ flu(bob). \\ hay\_fever(bob). \end{array}$$

The facts from disjoint-statements are distributed among the worlds as:

$$\begin{array}{lll} w_1 = & flu\_sneezing(bob). & w_2 = \quad null. \\ & hay\_fever\_sneezing(bob). & \quad hay\_fever\_sneezing(bob). \\ & P(w_1) = 0.7 \times 0.8 & \quad P(w_2) = 0.3 \times 0.8 \\ \\ w_3 = & flu\_sneezing(bob). & w_4 = \quad null. \\ & null. & \quad null. \\ & P(w_3) = 0.7 \times 0.2 & \quad P(w_4) = 0.3 \times 0.2 \end{array}$$

The query $sneezing(bob)$ *is true in three worlds and its probability is*

$$P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94.$$

Example 7. *The LPAD of Example 4 has four worlds* $\{w_1, w_2, w_3, w_4\}$*:*

$$\begin{array}{lll} w_1 = & sneezing(bob) \text{ :- } flu(bob). & w_2 = \quad null \text{ :- } flu(bob). \\ & sneezing(bob) \text{ :- } hay\_fever(bob). & \quad sneezing(bob) \text{ :- } hay\_fever(bob). \\ & flu(bob). & \quad flu(bob). \\ & hay\_fever(bob). & \quad hay\_fever(bob). \\ & P(w_1) = 0.7 \times 0.8 & \quad P(w_2) = 0.3 \times 0.8 \\ \\ w_3 = & sneezing(bob) \text{ :- } flu(bob). & w_4 = \quad null \text{ :- } flu(bob). \\ & null \text{ :- } hay\_fever(bob). & \quad null \text{ :- } hay\_fever(bob). \\ & flu(bob). & \quad flu(bob). \\ & hay\_fever(bob). & \quad hay\_fever(bob). \\ & P(w_3) = 0.7 \times 0.2 & \quad P(w_4) = 0.3 \times 0.2 \end{array}$$

The query $sneezing(bob)$ *is true in 3 worlds and its probability is*

$$P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$$

The probability of *sneezing(bob)* is calculated in a similar manner for PRISM and ProbLog.

Example 8. *PHA/ICL, PRISM and LPADs can have probabilistic statements with more than two alternatives. For example, the LPAD*

$$\begin{aligned} C_1 = \;& strong\_sneezing(X) : 0.3 \vee moderate\_sneezing(X) : 0.5 \quad :- \\ & flu(X). \\ C_2 = \;& strong\_sneezing(X) : 0.2 \vee moderate\_sneezing(X) : 0.6 \quad :- \\ & hay\_fever(X). \\ C_3 = \;& flu(david). \\ C_4 = \;& hay_{fever}(david). \end{aligned}$$

encodes the fact that flu and hay fever can cause strong sneezing, moderate sneezing or no sneezing. The clauses contain an extra atom null in the head that receives the missing probability mass and that is left implicit for brevity.

1.3.2 Equivalence of Expressive Power

To show that all these languages have the same expressive power for stratified Datalog programs, we discuss transformations among probabilistic constructs from the various languages. The mapping between PHA/ICL and PRISM translates each PHA/ICL disjoint statement into a multi-switch declaration and vice-versa in the obvious way. The mapping from PHA/ICL and PRISM to LPADs translates each disjoint statement/multi-switch declaration into a disjunctive LPAD fact.

The translation from an LPAD into PHA/ICL (first shown in [Vennekens and Verbaeten 2003]) rewrites each clause C_i with v variables $\overline{X}$

$$H_1 : p_1 \vee \ldots \vee H_n : p_n \text{ :- } B.$$

into PHA/ICL by adding n new predicates $\{choice_{i,1}/v, \ldots, choice_{i,n}/v\}$ and a disjoint statement:

$$\begin{aligned} & H_1 \text{ :- } B, choice_{i,1}(\overline{X}). \\ & \vdots \\ & H_n \text{ :- } B, choice_{i,n}(\overline{X}). \\ \\ & disjoint([choice_{i,1}(\overline{X}) : p_1, \ldots, choice_{i,n}(\overline{X}) : p_n]). \end{aligned}$$

For instance, clause C_1 of the LPAD of Example 8 is translated to

$$\begin{aligned} & strong\_sneezing(X) \text{ :- } flu(X), choice_{1,1}(X). \\ & moderate\_sneezing(X) : 0.5 \text{ :- } flu(X), choice_{1,2}(X). \\ & disjoint([choice_{1,1}(X) : 0.3, choice_{1,2}(X) : 0.5, choice_{1,3} : 0.2]). \end{aligned}$$

where the clause $null$:- $flu(X), choice_{1,3}$. is omitted since null does not appear in the body of any clause. Finally, as shown in [De Raedt et al. 2008], to convert LPADs to ProbLog, each

clause C_i with v variables $\overline{X}$

$$H_1 : p_1 \vee \ldots \vee H_n : p_n \text{ :- } B.$$

is translated into ProbLog by adding $n-1$ probabilistic facts for predicates $\{f_{i,1}/v, \ldots, f_{i,n}/v\}$:

$$\begin{array}{l} H_1 \text{ :- } B, f_{i,1}(\overline{X}). \\ H_2 \text{ :- } B, not(f_{i,1}(\overline{X})), f_{i,2}(\overline{X}). \\ \vdots \\ H_n \text{ :- } B, not(f_{i,1}(\overline{X})), \ldots, not(f_{i,n-1}(\overline{X})). \\ \\ \pi_1 :: f_{i,1}(\overline{X}). \\ \vdots \\ \pi_{n-1} :: f_{i,n-1}(\overline{X}). \end{array}$$

where $\pi_1 = p_1$, $\pi_2 = \frac{p_2}{1-\pi_1}$, $\pi_3 = \frac{p_3}{(1-\pi_1)(1-\pi_2)}$, ... In general $\pi_i = \frac{p_i}{\prod_{j=1}^{i-1}(1-\pi_j)}$. Note that while the translation into ProbLog introduces negation, the introduced negation only involves probabilistic facts, and so the transformed program will have a two-valued model whenever the original program does.

For instance, clause C_1 of the LPAD of Example 8 is translated to

$$\begin{array}{l} strong\_sneezing(X) \text{ :- } flu(X), f_{1,1}(X). \\ moderate\_sneezing(X) : 0.5 \text{ :- } flu(X), not(f_{1,1}(X)), f_{1,2}(X). \\ 0.3 :: f_{1,1}(X). \\ 0.71428571428 :: f_{1,2}(X). \end{array}$$

1.3.2.1 Additional Examples

Example 9. *The following program encodes the Mendelian rules of inheritance of the color of pea plants [Blockeel 2004]. The color of a pea plant is determined by a gene that exists in two forms (alleles), purple,* p*, and white,* w*. Each plant has two alleles for the color gene that reside on a couple of chromosomes.* cg(X,N,A) *indicates that plant* X *has allele* A *on chromosome* N*. The facts of the program express that* c *is the offspring of* f *and* m *and that the alleles of* m *are* ww *and of* f *are* pw*. The disjunctive rules encode the fact that an offspring inherits the allele on chromosome 1 from the mother and the allele on chromosome 2 from the father. In particular, each allele of the parent has a probability of 50% of being transmitted. The definite clauses for* color *express the fact that the color of a plant is purple if at least one of the alleles is* p*, i.e., that the* p *allele is dominant.*

```
color(X,white):- cg(X,1,w),cg(X,2,w).
color(X,purple):- cg(X,_A,p).
```

cg(X,1,A):0.5 ∨ cg(X,1,B):0.5 :- mother(Y,X),cg(Y,1,A),cg(Y,2,B).
cg(X,2,A):0.5 ∨ cg(X,2,B):0.5 :- father(Y,X),cg(Y,1,A),cg(Y,2,B).

mother(m,c). father(f,c).
cg(m,1,w). cg(m,2,w). cg(f,1,p). cg(f,2,w).

Example 10. *An interesting application of PLP under the distribution semantics is the computation of the probability of a path between two nodes in a graph in which the presence of each edge is probabilistic:*

path(X,X).
path(X,Y) :- path(X,Z),edge(Z,Y).

edge(a,b):0.3. edge(b,c):0.2. edge(a,c):0.6.

This program, coded in ProbLog, was used in [De Raedt et al. 2007] for computing the probability that two biological concepts are related in the BIOMINE network [Sevon et al. 2006].

1.3.3 Distribution Semantics for Stratified Programs with Function Symbols

When a program contains functions symbols, there is the possibility that its grounding may be infinite. If so, there may be an uncountably infinite number of worlds, or equivalently, the number of atomic choices in a selection that defines a world may be infinite. In this case, the probability of each individual world is zero since it is the product of infinite numbers all smaller than one. So the semantics as defined in Section 1.3 is not well-defined. The distribution semantics with function symbols has been proposed for PRISM [Sato 1995] and ICL [Poole 1997] but is easily applicable also to the other languages discussed in Section 1.2. Before delving into the semantics, we first present a motivating example.

Example 11. *A Hidden Markov Model (HMM) is a graphical model that represents a time-dependent process with a state and an output symbol for every time point. The state at time t depends only on the state at the previous time point $t-1$ while the output symbol at time t depends only on the state at the same time t. Let us suppose that the initial time is 0 and the final time is T. HMMs, which are used in speech recognition and many other applications, are usually represented as in Figure 1.1.*

There are various specialized algorithm for computing the probability of an output, the state sequence that most likely gave a certain output and the values of the parameters. However the HMM of Figure 1.1 can also be easily encoded as a probabilistic logic program (in this case an LPAD):

hmm(S,O) :- hmm(q1,[],S,O).
hmm(end,S,S,[]).

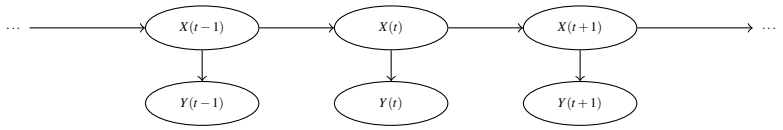


Figure 1.1 Hidden Markov Model.

```
hmm(Q,S0,S,[L|O]) :-
    Q \= end, next_state(Q,Q1,S0),
    emission(Q,L,S0), hmm(Q1,[Q|S0],S,O).

next_state(q1,q1,S):1/4 ∨ next_state(q1,q2,S):1/2 ∨
    next_state(q1,end,S):1/4.
next_state(q2,q1,S):1/2 ∨ next_state(q2,q2,S):1/4 ∨
    next_state(q2,end,S):1/4.

emission(q1,a,S):1/6 ∨ emission(q1,c,S):1/6 ∨
    emission(q1,g,S):1/6 ∨ emission(q1,t,S):1/2.
emission(q2,a,S):1/4 ∨ emission(q2,c,S):1/6 ∨
    emission(q2,g,S):5/12 ∨ emission(q2,t,S):1/6.
```

This programs models an HMM with three (hidden) states, q1, q2 and end, of which the last is an end state. The output symbols are a, c, g, and t. This HMM can for example model DNA sequences, where the output symbols are the amino-acids. The disjunctive clauses for next_state/3 define the transition probabilities, while the disjunctive clauses for emission/3 define the output probabilities. hmm(S,O) *is true when O is the output sequence for state sequence S. hmm*$(Q,S0,S,O)$ *is true when Q is the current state (time t), S*0 *is the list of previous states (up to time* $t-1$*), S is the final list of states (for time points in* $[0,T]$*) and O is the list of output symbols for the time interval* $[t,T]$*. next_state*$(Q,Q1,S)$ *is true when Q*1 *is the next state from current state Q and list of previous states S. emission*(Q,L,S) *is true when L is the output symbol from current state Q and list of previous states S.*

Note that the clauses for next_state$/3$ *and emission*$/3$ *have a variable argument holding the list of previous states. This is needed in order to have a different random variable for the next state and the output symbol for each time point. In fact, without that argument, in each world a single next state and output symbol would be always selected for each time point.*

Note that the above program uses a function symbol (the list constructor) for representing the sequence of visited states. While finite HMMs can be represented by Bayesian networks,

if a probabilistic logic program with functions has an infinite grounding or if it has cycles, then it cannot have a direct transformation into a Bayesian network. This theme will be discussed further in Section 1.5.3.

We now present the definition of the distribution semantics for programs with function symbols following [Poole 1997]. We preferred to follow [Poole 1997] because we think it is more constructive than [Sato 1995].

1.3.3.1 Algebras and Probability Measures

The semantics for a probabilistic logic program T with function symbols is given by defining a *probability measure* μ over the set of worlds W_T. Informally, μ assigns a probability to *subsets* of W_T, rather than to every element of W_T.[4] The approach dates back to [Kolmogorov 1950] who defined a probability measure μ as a real-valued function whose domain is a σ-*algebra* Ω of subsets of a set $\mathcal{W}$ called the *sample space*. Together $\langle \mathcal{W}, \Omega, \mu \rangle$ is called a *probability space*. [5]

Definition 1. *The set Ω of subsets of $\mathcal{W}$ is an* algebra *of $\mathcal{W}$ iff*

- *(a-1) $\mathcal{W} \in \Omega$;*
- *(a-2) Ω is closed under complementation,*
- *(a-3) Ω is closed under finite union, i.e., $\nu_1 \in \Omega, \nu_2 \in \Omega \rightarrow (\nu_1 \cup \nu_2) \in \Omega$*

The elements of Ω are called *measurable sets*. Importantly, for defining the distribution semantics for programs with function symbols, not every subset of $\mathcal{W}$ needs be present in Ω.

Definition 2. *Given a sample space $\mathcal{W}$ and an algebra Ω of subsets of $\mathcal{W}$, a (finitely additive)* probability measure *is a function $\mu : \Omega \rightarrow \mathbb{R}$ that satisfies the following axioms:*

- *(μ-1) $\mu(\nu) \geq 0$ for all $\nu \in \Omega$,*
- *(μ-2) $\mu(\mathcal{W}) = 1$;*
- *(μ-3) μ is finitely additive, i.e., if $O = \{\nu_1, \nu_2, \ldots\} \subseteq \Omega$ is a finite collection of pairwise disjoint sets, then $\mu(\bigcup_{\nu \in O}) = \sum_i \mu(\nu_i)$.*

1.3.3.2 Defining a Probability Measure for Programs with the Distribution Semantics

Towards defining a suitable algebra given a probabilistic logic program T, define the *set of worlds* ν_κ *compatible with* a finite composite choice κ as $\nu_\kappa = \{w_{\kappa'} \in W_T | \kappa \subseteq \kappa'\}$ where κ' is also a finite composite choice. Thus a composite choice identifies a set of worlds. For programs without function symbols $P(\kappa) = \sum_{w \in \nu_\kappa} P(w)$.

[4] The probability measure is then turned into a probability distribution by the identity map.

[5] For simplicity,we only consider the finite additivity version of probability spaces [Halpern 2003]. A more general case, the case of countable additivity, is detailed in [Sato and Kameya 2001]. In this case, probability measures are defined over σ-algebras.

Example 12. *For Example 1, consider* $\kappa = \{(C_1, \{X/bob\}, 1)\}$. *The set of worlds compatible with this composite choice is*

$flu\_sneezing(bob).$	$flu\_sneezing(bob).$
$hay\_fever\_sneezing(bob).$	$null.$
$P(w_1) = 0.7 \times 0.8$	$P(w_2) = 0.7 \times 0.2$

plus the non-probabilistic rules of Example 1. The probability of κ *is thus* $P(\kappa) = 0.7 = P(w_1) + P(w_2)$.

Given a *set* of composite choices K, the *set of worlds* ν_K *compatible with* K is $\nu_K = \bigcup_{\kappa \in K} \nu_\kappa$. Two composite choices κ_1 and κ_2 are *incompatible* if their union is not consistent. For example, the composite choices

$$\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$$

and

$$\kappa_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$$

are incompatible. A set K of composite choices is *pairwise incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K$, $\kappa_1 \neq \kappa_2$ implies that κ_1 and κ_2 are incompatible.

Note that in general, for programs without function symbols,

$$\sum_{\kappa \in K} P(\kappa) \neq \sum_{w \in \nu_K} P(w)$$

as can be seen from the following example.

Example 13. *The set of composite choices for Example 1*

$$K = \{\kappa_1, \kappa_2\} \tag{1.1}$$

with $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$ *and* $\kappa_2 = \{(C_2, \{X/bob\}, 1)\}$ *is such that* $P(\kappa_1) = 0.7$ *and* $P(\kappa_2) = 0.8$ *but* $\sum_{w \in \nu_K} P(w) = 0.94$.

If, on the other hand, K is pairwise incompatible then

$$\sum_{\kappa \in K} P(\kappa) = \sum_{w \in \nu_K} P(w).$$

For example, consider

$$K' = \{\kappa_1, \kappa_2'\} \tag{1.2}$$

with $\kappa_2' = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$. $P(\kappa_2') = 0.3 \cdot 0.8 = 0.24$ *so the property holds with the probabilities of the worlds summing up to 0.94.*

Regardless of whether a probabilistic logic program has a finite number of worlds or not, obtaining pairwise incompatible sets of composite choices is an important problem because

one way to assign probabilities to a set K of composite choices is to construct an equivalent set that is pairwise incompatible. Two sets K_1 and K_2 of finite composite choices are *equivalent* if they correspond to the same set of worlds: $\nu_{K_1} = \nu_{K_2}$.

Then the *probability of a pairwise incompatible set K of composite choices* is defined as

$$P(K) = \sum_{\kappa \in K} P(\kappa). \tag{1.3}$$

Given a set K of composite choices, an equivalent set that is pairwise incompatible can be constructed through the technique of *splitting*. More specifically, if $F\theta$ is an instantiated formula and κ is a composite choice that does not contain an atomic choice (F, θ, i) for any o, the *split* of κ on $F\theta$ is the set of composite choices

$$S_{\kappa, F\theta} = \{\kappa \cup \{(F, \theta, 1)\}, \ldots, \kappa \cup \{(F, \theta, n)\}\}$$

where n is the number of alternatives in F. It is easy to see that κ and $S_{\kappa,F\theta}$ identify the same set of possible worlds, i.e., that $\nu_\kappa = \nu_{S_{\kappa,F\theta}}$. For example, the split of $\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$ on $C_2\{X/bob\}$ is

$$\{\{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\}, \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 2)\}\}$$

The technique of splitting composite choices on formulas is used for the following result [Poole 2000].

Theorem 1 (Existence of a pairwise incompatible set of composite choices [Poole 2000])**.** *Given a finite set K of composite choices, there exists a finite set K' of pairwise incompatible composite choices such that K and K' are equivalent.*

Proof. Given a finite set of composite choices K, there are two possibilities to form a new set K' of composite choices so that K and K' are equivalent:

1. **removing dominated elements**: if $\kappa_1, \kappa_2 \in K$ and $\kappa_1 \subset \kappa_2$, let $K' = K \setminus \{\kappa_2\}$.
2. **splitting elements**: if $\kappa_1, \kappa_2 \in K$ are compatible (and neither is a superset of the other), there is a $(F, \theta, i) \in \kappa_1 \setminus \kappa_2$. We replace κ_2 by the split of κ_2 on $F\theta$. Let $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2, F\theta}$.

In both cases $\nu_K = \nu_{K'}$. If we repeat this two operations until neither is applicable, we obtain a splitting algorithm (see Figure 1.2) that terminates because K is a finite set of composite choices. The resulting set K' is pairwise incompatible and is equivalent to the original set. For example, the splitting algorithm applied to K (1.1) can return K' (1.2). □

Theorem 2 (Equivalence of the probability of two equivalent pairwise incompatible finite sets of finite composite choices [Poole 1993a])**.** *If K_1 and K_2 are both pairwise incompatible finite sets of finite composite choices such that they are equivalent then $P(K_1) = P(K_2)$.*

```
1: procedure SPLIT(K)
2:     Input: set of composite choices K
3:     Output: pairwise incompatible set of composite choices equivalent to K
4:     loop
5:         if ∃κ1, κ2 ∈ K and κ1 ⊂ κ2 then
6:             K ← K \ {κ2}
7:         else
8:             if ∃κ1, κ2 ∈ K compatible then
9:                 choose (F, θ, i) ∈ κ1 \ κ2
10:                K ← K \ {κ2} ∪ S_{κ2,Fθ}
11:            else
12:                exit and return K
13:            end if
14:        end if
15:    end loop
16: end procedure
```

Figure 1.2 Splitting Algorithm.

Proof. Consider the set D of all instantiated formulas $F\theta$ that appear in an atomic choice in either K_1 or K_2. This set is finite. Each composite choice in K_1 and K_2 has atomic choices for a subset of D. For both K_1 and K_2, we repeatedly replace each composite choice κ of K_1 and K_2 with its split $S_{\kappa,F_i\theta_j}$ on an $F_i\theta_j$ from D that does not appear in κ. This procedure does not change the total probability as the probabilities of $(F_i,\theta_j,1),\ldots,(F_i,\theta_j,n)$ sum to 1.

At the end of this procedure the two sets of composite choices will be identical. In fact, any difference can be extended into a possible world belonging to ν_{K_1} but not to ν_{K_2} or vice versa. □

Example 14. *Recall from Example 13 the set of composite choices* $K' = \{\kappa_1, \kappa_2'\}$ *with*

$$\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$$

and

$$\kappa_2' = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}.$$

Consider also the composite choices

$$\kappa_{1.1}' = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 1)\},$$

$$\kappa_{1.2}' = \{(C_1, \{X/bob\}, 1), (C_2, \{X/bob\}, 2)\}$$

and the set $K'' = \{\kappa'_{1.1}, \kappa'_{1.2}, \kappa'_2\}$. *Note that* K' *and* K'' *are equivalent and are both pairwise incompatible. By Theorem 2 their probabilities are equivalent:*

$$P(K') = 0.7 + 0.3 \times 0.8 = 0.94$$

while

$$P(K'') = 0.7 \times 0.8 + 0.7 \times 0.2 + 0.3 \times 0.8 = 0.94.$$

For a probabilistic logic program T, we can thus define a unique probability measure $\mu : \Omega_T \to [0,1]$ where Ω_T is defined as the set of sets of worlds identified by finite sets of finite composite choices:

$$\Omega_T = \{\nu_K | K \text{ is a finite set of finite composite choices}\}.$$

The corresponding measure μ is defined by $\mu(\nu_K) = P(K')$ where K' is a pairwise incompatible set of composite choices equivalent to K.

Theorem 3. $\langle W_T, \Omega_T, \mu \rangle$ *is a finitely additive probability space.*

Proof. Ω_T is an algebra over W_T since $W_T = \nu_K$ with $K = \{\emptyset\}$. Moreover, the complement of ν_K where K is a finite set of finite composite choice is $\nu_{\overline{K}}$ where $\overline{K}$ is a certain finite set of finite composite choice. In fact, $\overline{K}$ can be obtained with the function *duals*(K) of [Poole 2000] that performs Reiter's hitting set algorithm over K, generating an element κ of $\overline{K}$ by picking an atomic choice (C, θ, k) from each element of K and inserting in κ an incompatible atomic choice, i.e., an atomic choice (C, θ, k') with $k \neq k'$. After this process is performed in all possible ways, inconsistent sets of atom choices are removed obtaining $\overline{K}$. Since the possible choices of the atomic choices and of their incompatible counterparts is finite, so is $\overline{K}$.

Finally, condition (a-3) holds since the union of ν_{K_1} with ν_{K_2} is equal to $\nu_{K_1 \cup K_2}$ by definition.

μ is a probability measure because $\mu(\nu_{\{\emptyset\}}) = 1$, $\mu(\nu_K) \geq 0$ for all K and if $\nu_{K_1} \cap \nu_{K_2} = \emptyset$ and K'_1 (K'_2) is pairwise incompatible and equivalent to K_1 (K_2), then $K'_1 \cup K'_2$ is pairwise incompatible because $\nu_{K_1} \cap \nu_{K_2} = \emptyset$ and

$$\mu(\nu_{K_1} \cup \nu_{K_2}) = \sum_{\kappa \in K'_1 \cup K'_2} P(\kappa) = \sum_{\kappa_1 \in K'_1} P(\kappa_1) + \sum_{\kappa_2 \in K'_2} P(\kappa_2) = \mu(\nu_{K_1}) + \mu(\nu_{K_2}).$$

□

Given a query Q, a composite choice κ is an *explanation* for Q if

$$\forall w \in \nu_\kappa, \;\; w \models Q$$

A set K of composite choices is *covering* wrt Q if every world in which Q is true belongs to ν_K

Definition 3. *For a probabilistic logic program T, the probability of a ground atom Q is given by*

$$P(Q) = \mu(\{w | w \in W_T, w \models Q\})$$

If Q has a finite set K of finite explanations such that K is covering then $\{w | w \in W_T \wedge w \models Q\} = \nu_K \in \Omega_T$ and we say that $P(Q)$ is *well-defined* for the distribution semantics. A program T is well-defined if the probability of all ground atoms in the grounding of T is well-defined.

Example 15. *Consider the PHA/ICL program of Example 1. The two composite choices:*

$$\kappa_1 = \{(C_1, \{X/bob\}, 1)\}$$

and

$$\kappa_2 = \{(C_1, \{X/bob\}, 2), (C_2, \{X/bob\}, 1)\}$$

are such that $K = \{\kappa_1, \kappa_2\}$ is a pairwise incompatible finite set of finite explanations that are covering for the query $Q = sneezing(bob)$. Definition 3 therefore applies, and $P(Q) = P(\kappa_1) + P(\kappa_2) = 0.7 + 0.3 \cdot 0.8 = 0.94$

1.3.3.3 Comparison with Sato and Kameya's Definition

[Sato and Kameya 2001] build a probability measure on the sample space W_T from a collection of finite distributions. Let $T' = \{C_1, C_2, \ldots\}$ be the grounding of T and let X_i be a random variable associated to C_i whose domain is $\{1, \ldots, j_i\}$ where j_i is the number of alternatives of C_i.

The finite distributions $P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n)$ for $n \geq 1$ must be such that

$$\begin{cases} 0 \leq P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n) \leq 1 \\ \sum_{k_1, \ldots, k_n} P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n) = 1 \\ \sum_{k_{n+1}} P_T^{(n+1)}(X_1 = k_1, \ldots, X_{n+1} = k_{n+1}) = \\ \quad P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n) \end{cases} \tag{1.4}$$

where $k_i \in \{1, \ldots, j_i\}$. The last equation is called the compatibility condition. It can be proved [Chow and Teicher 2012] from the compatibility condition that there exists a probability space (W_T, Ψ_T, η) where η is a probability measure on Ψ_T, the minimal σ-algebra containing open sets of W_T such that for any n,

$$\eta(X_1 = k_1, \ldots, X_n = k_n) = P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n). \tag{1.5}$$

[Sato and Kameya 2001] define $P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n)$ as $P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n) = p_1 \ldots p_n$ where p_i is the annotation of alternative k_i in clause C_i. This definition clearly satisfies the properties in (1.4).

It can be shown that this definition of the distribution semantics with function symbols coincides with the one given above following [Poole 1997] in the case that each ground atom has a finite set of finite explanations that is covering.

In this case, in fact, $X_1 = k_1, \ldots, X_n = k_n$ is equivalent to the set of composite choices $K = \{\{(C_1, \emptyset, k_1), \ldots, (C_n, \emptyset, k_n)\}\}$ and $\mu(\nu_K)$ is equal to $p_1 \ldots p_n$, which satisfies equation (1.5).

1.3.3.4 Programs for which the Distribution Semantics is Well-defined

Inference procedures often rely on the computation of a covering set of explanations. In this sense, the notion of well-defined programs is important, as it ensures that the set of explanations is finite and so is each explanation. Therefore an important open question is to understand when a program or query is well-defined, and to identify cases where there are decidable algorithms to determine this. For instance, ground queries to the program of Example 11, which describe an HMM, are well-defined as each such query has a finite set of finite explanations which is covering – even though there are an infinite number of such queries.

In PRISM well-definedness of a program is explicitly required [Sato and Kameya 2001]. In PHA/ ICL the program (excluding disjoint statements) is required to be acyclic [Apt and Bezem 1991]. The condition of modular acyclicity is proposed in [Riguzzi 2009] to enlarge the set of programs. This condition was weakened in [Riguzzi and Swift 2013] to the set of programs that are bounded term-size, a property whose definition is based on dynamic stratification. While the property of being bounded term-size is semi-decidable, such programs include a number of recent static classes for programs with finite models (cf. [Alviano et al. 2010, Baselice and Bonatti 2010, Calimeri et al. 2011, Greco et al. 2013] for some recent work on decidability of stable models).

The works [Gorlin et al. 2012, Sato and Meyer 2012] go beyond well-definedness by presenting inference algorithm that can deal with infinite explanations for restricted classes of programs.

1.3.4 The Distribution Semantics for Non-Stratified Programs

1.3.4.1 The Well-Founded Semantics

The distribution semantics can be extended in a straightforward manner to the well-founded semantics (WFS)[6]. In the following, $w_\sigma \models L$ means that the ground *literal* L is true in the well-founded model of the program w_σ.

For a literal L_j, let $t(L_j)$ stand as shorthand for $L_j = true$. We extend the probability distribution on programs to ground literals by assuming $P(t(L_j)|w) = 1$ if L_j is true in w and 0 otherwise (i.e., if L_j is false or undefined in w). Thus the probability of L_j being true in

[6] As an alternative approach, [Sato et al. 2005] provides a semantics for negation in probabilistic programs based on the three-valued Fitting semantics for logic programs.

a program T without function symbols is

$$P(t(L_j)) = \sum_{w \in W_T} P(t(L_j), w) = \sum_{w \in W_T} P(t(L_j)|w)P(w) = \sum_{w \in W_T, w \models L_j} P(w).$$

Example 16. *The barber paradox, introduced by Bertrand Russell [Russell 1967], is expressed as:*

The village barber shaves everyone in the village who does not shave himself.

The paradox was modeled as a logic program under WFS in [Dung 1991]. Making things probabilistic, the paradox can be modeled as the LPAD:

shaves(barber,Person):- villager(Person),not shaves(Person,Person).
C_1 shaves(barber,barber):0.25.
C_2 shaves(doctor,doctor):0.25.

villager(barber). villager(mayor). villager(doctor).

where the facts that the barber and the doctor shave theeemselves are probabilistic.
There are four different worlds associated with this LPAD.

- w_1*: both* C_1 *and* C_2 *are selected. In this world*
  shaves(barber,barber), shaves(barber,mayor) *and* shaves(doctor,doctor)
  are all true. The probability of w_1 *is* $\frac{1}{16}$.
- w_2*:* C_1 *is selected but not* C_2*. In this world*
  shaves(barber,barber) shaves(barber,mayor) *and* shaves(barber,doctor)
  are all true. The probability of w_2 *is* $\frac{3}{16}$.
- w_3*:* C_2 *is selected but not* C_1*. In this world*
  shaves(barber,mayor) *and* shaves(doctor,doctor)
  are true, while shaves(barber,barber) *is undefined. The probability of* w_3 *is* $\frac{3}{16}$.
- w_4*: neither* C_1 *nor* C_2 *is selected. In this world*
  shaves(barber,mayor) *and* shaves(barber,doctor)
  are true, while shaves(barber,barber) *is undefined. The probability of* w_4 *is* $\frac{9}{16}$.

In each of the above world, each ground instantiation of shaves/2 *that is not explicitly mentioned is false.*
Given the probabilities of each world, the probability of each literal can be computed:

- $P(shaves(doctor, doctor)) = P(w_1) + P(w_3) = \frac{1}{4}$;
  - $P(not\ shaves(doctor, doctor)) = P(w_2) + P(w_4) = \frac{3}{4}$;
- $P(shaves(barber, doctor)) = P(w_2) + P(w_4) = \frac{3}{4}$;
  - $P(not\ shaves(barber, doctor)) = P(w_1) + P(w_3) = \frac{1}{4}$;
- $P(shaves(barber, barber)) = P(w_1) + P(w_2) = \frac{1}{4}$;

- $P(not\ shaves(barber, barber) = 0$

Note that $P(A) = 1 - P(not\ A)$*, except for the case where A is* shaves(barber, barber).

From the perspective of modeling, the use of the well-founded semantics provides an approximation of the probability of an atom and of its negation, and thus may prove useful for domains in which a cautious under-approximation of probabilities is necessary. In addition, as discussed in Section 1.6.4, using the third truth value of the well-founded semantics offers a promising approach to semantically sound approximation of probabilistic inference.

1.3.4.2 The Stable Model Semantics

P-log [Baral et al. 2009] is a formalism for introducing probability in Answer Set Programming (ASP). P-log has a rich syntax that allows expression of a variety of stochastic and non-monotonic information. The semantics of a P-log program T is given in terms of its translation into an Answer Set program $\pi(T)$ whose stable models are the possible worlds of T. The following simple program from [Baral et al. 2009] illustrates certain aspects of P-log.

Example 17. *Consider two gamblers,* john *and* mike*:* john *owns a die that is fair, while* mike *owns a die that isn't. This situation can be represented by the P-log program below. The first portion of the program is a declaration of* sorts*, such as* dice*,* score *and* person *along with* attributes *such as* roll *and* owner*. These declarations allow P-log syntax to extend ASP syntax to include (possibly non-ground)* attribute terms *such as* roll(Die) *along with* atomic statements *such as* roll(Die) = 6*. In addition to rules with this extended syntax, P-log has* random selection *rules to indicate that certain attributes may be considered random over a certain domain. A simple example of such a rule is* random(roll(Die))*, which indicates, since* Die *is not restricted, that* roll *is random attribute over the entire domain dice. P-log also contains* probability atoms *indicating the probabilities of atomic statements (e.g.,* pr(roll(Die) = Score | owner(Die) = john) = $\frac{1}{6}$*) which indicates that if the owner of a given die,* Die*, is* john*, the probability of* roll(Die) *is* $\frac{1}{6}$.

Declaration

dice = {d1,d2}. score = {1,2,3,4,5,6}. roll: dice → score.
person = {mike,john}. owner: dice → person,

Rules

owner(d1) = john. owner(d2) = mike.

Random Selection

random(roll(Die)).

Probabilistic Information

pr(roll(Die) = Score | owner(Die) = john) = $\frac{1}{6}$.
pr(roll(Die) = 6 | owner(Die) = mike) = $\frac{1}{4}$.
pr(roll(Die) = Score | Score ≠ 6,owner(Die) = mike) = $\frac{3}{20}$.

In order to evaluate the above program, the declarations are used to translate the rules and random selections into a standard ASP program. Atomic statements are translated to ground atoms using the sorts of the attribute domains and ranges if necessary: for instance owner(d1) = john *is translated to the atom* owner(d1,john). *In a similar manner, random selection rules are translated to ground disjunctions (in ASP, a disjunction in a fact or rule head is taken as an exclusive use of "or"). In the program above, the random selection* random(roll(Die)) *is translated into two disjunctive ground facts:*

roll(d1,1) $\vee$ roll(d1,2) $\vee$ roll(d1,3) $\vee$ roll(d1,4) $\vee$ roll(d1,5) $\vee$ roll(d1,6).

and

roll(d2,1) $\vee$ roll(d2,2) $\vee$ roll(d2,3) $\vee$ roll(d2,4) $\vee$ roll(d2,5) $\vee$ roll(d2,6).

As shown in the example above, random selection rules correspond to probabilistic constructs, so that when a P-log program T is translated into an ASP program $\pi(T)$ the stable models of $\pi(T)$ will contain total composite choices and so will correspond to possible worlds. The probability for each stable model $M_{\pi(T)}$ is constructed using the probability atoms that are satisfied in $M_{\pi(T)}$. A probability is then assigned to each stable model; the probability of a query Q is given, as for other distribution semantics languages, by the sum of the probabilities of the possible worlds where Q is true. CONTRADICTIONS

Example 18 (Example 16 Continued)**.** *There are 36 stable models for* $\pi(T)$ *of the previous example: one for each score for* d1 *and* d2*. The probability of each world containing* roll(d2,6) *is* $\frac{1}{24}$ *while the probability of each other world is* $\frac{1}{40}$

P-log differs from the languages mentioned in Section 1.2 because the possible worlds are generated not only because of stochastic choices but also because of disjunctions and non-stratified negations appearing in the logical part of a P-log program. As a consequence, the distribution obtained by multiplying all the probability factors of choices that are true in a stable model is not normalized. In order to get a probability distribution over possible worlds, the unnormalized probability of each stable model must be divided by the sum of the unnormalized probabilities of each possible world.

In most of the literature on P-log, function symbols are not handled by the semantics, although [Gelfond and Rushton 2010] provides recent work towards this end. Furthermore, recent work that extends stable models to allow function symbols [Alviano et al. 2010, Baselice and Bonatti 2010, Calimeri et al. 2011, Greco et al. 2013] may also lead to finite well-definedness conditions for P-log programs containing function symbols.

1.4 Other Semantics for Probabilistic Logics

Here we briefly discuss a few examples of frameworks related to probabilistic logic programming that are outside of the distribution semantics. Our goal in this section is simply to give

the flavor of other possible approaches; a complete accounting of such frameworks is beyond the scope of this chapter.

1.4.1 Stochastic Logic Programs

Stochastic Logic Programs (SLPs) [Cussens 2001, Muggleton 1996] are logic programs with parameterized clauses which define a distribution over refutations of goals. The distribution provides, by marginalisation, a distribution over variable bindings for the query. SLPs are a generalization of stochastic grammars and hidden Markov models.

An *SLP S* is a definite logic program where some of the clauses are of the form $p : C$ where $p \in \mathbb{R}, p \geq 0$ and C is a definite clause. Let $n(S)$ be the definite logic program obtained by removing the probability labels. A *pure* SLP is an SLP where all clauses have probability labels. A *normalized* SLP is one where probability labels for clauses whose heads share the same predicate symbol sum to one.

In pure SLPs each SLD derivation for a query Q is assigned a real label by multiplying the labels of each individual derivation step. The label of a derivation step where the selected atom unifies with the head of clause $p_i : C_i$ is p_i. The probability of a successful derivation from Q is the label of the derivation divided by the sum of the labels of all the successful derivations. This clearly is a distribution over successful derivations from Q

The probability of an instantiation $Q\theta$ is the sum of the probabilities of the successful derivations that produce $Q\theta$. It can be shown that the probabilities of all the atoms for a predicate q that succeed in $n(S)$ sum to one, i.e., S defines a probability distribution over the success set of q in $n(S)$.

In impure SLPs, the unparameterized clauses are seen as non-probabilistic domain knowledge acting as constraints. To this purpose, derivations are identified with the set of the parameterized clauses they use. In this way, derivations that differ only on the unparameterized clauses form an equivalence class.

Given their similarity with stochastic grammars and hidden Markov models, SLPs are particularly suited to represent this kind of models. They differ from the distribution semantics because they define a probability distribution over instantiations of the query, while the distribution semantics defines a distribution over the truth values of ground atoms.

1.4.2 Nilsson's probabilistic logic

Nilsson's probabilistic logic [Nilsson 1986] takes an approach different from the distribution semantics for combining logic and probability: while the first considers sets of distributions, the latter computes a single distribution over possible worlds. In Nilsson's logic, a probabilistic interpretation Pr defines a probability distribution over the set of interpretations Int. The probability of a logical formula F according to Pr, denoted $Pr(F)$, is the sum of all $Pr(I)$ such that $I \in Int$ and $I \models F$. A probabilistic knowledge base $\mathcal{W}$ is a set of probabilistic formulas of the form $F \geq p$. A probabilistic interpretation Pr satisfies $F \geq p$ iff $Pr(F) \geq p$. Pr satisfies

$\mathcal{W}$, or Pr is a model of $\mathcal{W}$, iff Pr satisfies all $F \geq p \in \mathcal{W}$. $Pr(F) \geq p$ is a tight logical consequence of $\mathcal{W}$ iff p is the infimum of $Pr(F)$ in the set of all models Pr of $\mathcal{W}$. Computing tight logical consequences from probabilistic knowledge bases can be done by solving a linear optimization problem.

Nilsson's logic allows different consequences to be drawn from logical formulas than the distribution semantics. Consider a ProbLog program (cf. Section 1.2) composed of the facts $0.4 :: c(a).$ and $0.5 :: c(b).$; and a probabilistic knowledge base composed of $c(a) \geq 0.4$ and $c(b) \geq 0.5$. For the distribution semantics $P(c(a) \vee c(b)) = 0.7$, while with Nilsson's logic the lowest p such that $Pr(c(a) \vee c(b)) \geq p$ holds is 0.5. This difference is due to the fact that, while in Nilsson's logic no assumption about the independence of the statements is made, in the distribution semantics the probabilistic axioms are considered as independent. While independencies can be encoded in Nilsson's logic by carefully choosing the values of the parameters, reading off the independencies from the theories becomes more difficult.

The assumption of independence of probabilistic axioms does not restrict expressiveness as one can specify any joint probability distribution over the logical ground atoms, possibly introducing new atoms if needed. This claim is substantiated by the fact that Bayesian networks can be encoded in probabilistic logic programs under the distribution semantics, as discussed in Section 1.5.3.

1.4.3 Markov Logic Networks

A Markov Logic Network (MLN) is a first order logical theory in which each sentence has a real-valued weight. An MLN is a template for generating Markov networks, graphical models where the edges among variables are undirected. Given sets of constants defining the domains of the logical variables, an MLN defines a Markov network that has a node for each ground atom and edges connecting the atoms appearing together in a grounding of a formula. MLNs follow the so-called Knowledge Base Model Construction approach for defining a probabilistic model [Bacchus 1993, Wellman et al. 1992] in which the probabilistic-logic theory is a template for generating an underlying probabilistic graphical model (Bayesian or Markov networks). The probability distribution encoded by an MLN is

$$P(\mathbf{x}) = \frac{1}{Z} \exp\Big(\sum_{f_i \in M} w_i n_i(\mathbf{x}) \Big)$$

where $\mathbf{x}$ is a joint assignment of truth value to all atoms in the Herbrand base, M is the model, f_i is the i-th formula in M, w_i is its weight, $n_i(\mathbf{x})$ is the number of groundings of formula f_i that are satisfied in $\mathbf{x}$ and Z is a normalization constant.

A probabilistic logic program T under the distribution semantics differs from an MLN because T has a semantics defined directly rather than through graphical models (though there are strong relationships to graphical models, see Section 1.5) and because restricting the

logic of choice to be logic programming, rather than full first-order logic, permits to exploit the plethora of techniques developed in logic programming.

1.4.4 Evidential Probability

Evidential probability (cf. [Kyburg and Teng 2001]) is an approach to reason about probabilistic information that may be approximate, incomplete or even contradictory. Evidential probability adds statistical statements of the form

$$\%\overline{Vars}(Target, Reference, Lower, Upper) \tag{1.6}$$

where *Target* and *Reference* are formulas, and *Lower* and $Upper$ are numbers between 0 and 1. Although the syntax is unusual, it may help to think of % as a quantifier and $\overline{Vars}$ as the set of variables open in the *Target* and *Reference* formulas. The statistical statment thus states that the proportion of instantiations of $\overline{Vars}$ for which *Target* is true, among those for which *Reference* is also true, is between *Lower* and $Upper$.

As an example, consider

$$\%X(in\_urn(u_1, X), is\_blue(X), 0.3, 0.4)$$

which can be read as *The percentage of X such that in_urn(u_1,X) is true, where is_blue(X) also true, is between 0.3 and 0.4.* This illustrates and important special case that occurs when *Vars* is a singleton variable shared by *Target* and *Reference*. In this case the statement can be read as one of probabilistic set membership of a given individual in a domain, and we restrict the rest of this discussion to this special case.

To see how this works in practice consider the problem of determining the likelihood of whether an individual o_1 is in a class C (when o_1 cannot be proved for certain to be in C). Each statistical statement $S = \%X(C_T, C_R, L, U)$ is collected for which o_1 is known to be an element of the target class C_T of S and for which C is a superset of the reference class of C_R (i.e., C_R implies C). For instance, in the above example, the reference class indicates the uncertain property of whether a ball were blue, so for a given ball, target classes of applicable statistical statements indicate properties known about the ball, while reference classes of these statements consist of properties that imply blueness. Once the applicable statements have been identified, a series of rules is used to derive a single interval from these collected statements, and to weigh the evidence provided for different statements if their intervals are contradictory. (Two intervals contradict each other if neither is a subinterval of the other.) One such rule is the principle of *specificity*: a statement S_1 may override statement S_2 if the target class of S_1 is more specific to o_1 than that of S_2. (For instance, a statistical statement about an age cohort might override a statement about the general population.)

Evidential probability is thus not a probabilistic logic, but a meta-logic for defeasible reasoning about statistical statements once non-probabilistic aspects of a model have been de-

rived. It is thus less powerful than probabilistic logics based on the distribution semantics, but is applicable to situations where such logics don't apply, due to contradiction, incompleteness, or other factors.

1.4.5 Annotated Probabilistic Logic Programs

Another approach is that of Annotated Probabilistic Logic Programming (Annotated PLP) [Ng and Subrahmanian 1992], which allows program atoms to be annotated with intervals that can be interpreted probabilistically. An example rule in this approach:

$$a : [0.75, 0.85] \leftarrow b : [1, 1], c : [0.5, 0.75]$$

can be taken as stating that the probability of a is between 0.75 and 0.85 if b is certainly true and the probability of c is between 0.5 and 0.75. The probability interval of a conjunction or disjunction of atoms is defined using a combinator to construct the tightest bounds for the formula. For instance if d is annotated with $[l_d, h_d]$ and e with $[l_e, h_e]$ the probability of $a \wedge b$ is annotated with

$$[max(0, l_d + l_e - 1), min(h_d, h_e)].$$

Using these combinators, an inference operator and fixed point semantics is defined for positive Datalog programs. A model theory is obtained for such programs by considering the annotations as constraints on acceptable probabilistic worlds: an Annotated PLP thus describes a family of probabilistic worlds.

Annotated PLPs have the advantage that deduction is of low complexity, as the logic is truth-functional, i.e., the probability of a query can be computed directly using combinators. The corresponding disadvantages are that Annotated PLPs may be inconsistent if they are not carefully written, and that the use of the above combinators may quickly lead to assigning overly slack probability intervals to certain atoms. These aspects are partially addressed by Hybrid Annotated PLPs [Dekhtyar and Subrahmanian 2000], which allow different flavors of combinators based on e.g., independence or mutual exclusivity of given atoms.

1.5 Probabilistic Logic Programs and Bayesian Networks

In this section, we first present two examples of probabilistic logic programs whose semantics is explicitly related to Bayesian Networks: Bayesian Logic Programs and Knowledge Base Model Construction. Making use of the formalism of Bayesian Logic Programs, we then discuss the relationship of Bayesian Networks to the distribution semantics (for background on Bayesian networks cf. [Pearl 1988] or similar texts).

1.5.1 Bayesian Logic Programs

Bayesian Logic Programs (BLPs) [Kersting and Raedt 2001] use logic programming to compactly encode a large Bayesian network. In BLPs, each ground atom represents a random

variable and the clauses define the dependencies between ground atoms. A clause of the form

$$A|A_1,\ldots,A_m$$

indicates that, for each of its groundings $(A|A_1,\ldots,A_m)\theta$, $A\theta$ has $A_1\theta$, ..., $A_m\theta$ as parents. The domains and conditional probability tables (CPTs) for the ground atom/random variables are defined in a separate portion of the model. In the case where a ground atom $A\theta$ appears in the head of more than one clause, a *combining rule* is used to obtain the overall CPT from those given by individual clauses.

For example, in the Mendelian genetics program of Example 9, the dependency that gives the value of the color gene on chromosome 1 of a plant as a function of the color genes of its mother can be expressed as

cg(X,1)|mother(Y,X),cg(Y,1),cg(Y,2).

where the domain of atoms built on predicate *cg/2* is $\{p,w\}$ and the domain of *mother(Y,X)* is Boolean. A suitable CPT should then be defined that assigns equal probability to the alleles of the mother to be inherited by the plant.

1.5.2 Knowledge Base Model Construction

In Knowledge Base Model Construction (KBMC) [Bacchus 1993, Wellman et al. 1992], PLP is a template for building a complex Bayesian network. The semantics of the probabilistic logic program is then given by the semantics of the generated network. For example, in a CLP(BN) program [Costa et al. 2003], logical variables can be random. Their domain, parents and CPTs are defined by the program. Probabilistic dependencies are expressed by means of CLP constraints:

{ *Var = Function with p(Values, Dist)* }
{ *Var = Function with p(Values, Dist, Parents)* }

The first form indicates that the logical variable *Var* is random with domain *Values* and CPT *Dist* but without parents; the second form defines a random variable with parents. In both forms, *Function* is a term over logical variables that is used to parameterize the random variable: a different random variable is defined for each instantiation of the logical variables in the term. For example, the following snippet from a school domain:

course_difficulty(CKey, Dif) :-
    { *Dif = difficulty(CKey) with p([h,m,l], [0.25, 0.50, 0.25])* }.

defines the random variable *Dif* with values *h*, *m* and *l* representing the difficulty of the course identified by *CKey*. There is a different random variable for every instantiation of *CKey* — i.e., for each course. In a similar manner, the intelligence *Int* of a student identified by *SKey* is given by

```
student_intelligence(SKey, Int) :-
        { Int = intelligence(SKey) with p([h, m, l], [0.5,0.4,0.1]) }.
```

Using the above predicates, the following snippet predicts the grade received by a student when taking the exam of a course.

```
registration_grade(Key, Grade) :-
        registration(Key, CKey, SKey),
        course_difficulty(CKey, Dif),
        student_intelligence(SKey, Int),
        { Grade = grade(Key) with p(['A','B','C','D'],
            %h/h h/m h/l m/h m/m m/l  l/h  l/m  l/l
            [0.20,0.70,0.85,0.10,0.20,0.50,0.01,0.05,0.10,   % 'A'
             0.60,0.25,0.12,0.30,0.60,0.35,0.04,0.15,0.40,   % 'B'
             0.15,0.04,0.02,0.40,0.15,0.12,0.50,0.60,0.40,   % 'C'
             0.05,0.01,0.01,0.20,0.05,0.03,0.45,0.20,0.10 ], % 'D'
            [Int,Dif]) }.
```

Here *Grade* indicates a random variable parameterized by the identifier *Key* of a registration of a student to a course. The code states that there is a different random variable *Grade* for each student's registration in a course and each such random variable has possible values *'A'*, *'B'*, *'C'* and *'D'*. The actual value of the random variable depends on the intelligence of the student and on the difficulty of the course, that are thus its parents. Together with facts for *registration/3* such as

```
registration(r0,c16,s0).        registration(r1,c10,s0).
registration(r2,c57,s0).        registration(r3,c22,s1).
....
```

the code defines a Bayesian network with a *Grade* random variable for each registration. CLP(BN) is implemented as a library of YAP Prolog [Costa et al. 2012].

1.5.3 Conversion of PLP under the Distribution Semantics to Bayesian Networks

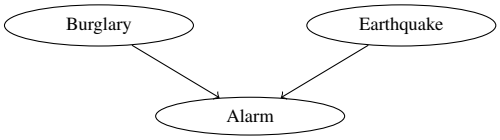
In [Vennekens and Verbaeten 2003] the relationship between LPADs and BLPs is investigated in detail. The authors show that ground BLPs can be converted to ground LPADs and that ground acyclic LPADs can be converted to ground BLPs. [7] A logic program is *acyclic* [Apt and Bezem 1991] if its atom dependency graph is acyclic. As a BLP directly encodes a Bayesian network, the results of [Vennekens and Verbaeten 2003] allow us to draw a connection between LPADs and Bayesian networks.

[7] We note that this equivalence is only for ground programs, unlike the equivalences of Section 1.3.2 which hold for non-ground programs.

Example 19. *Figure 1.3 shows a simple Bayesian network for reasoning about the causes of a building alarm. This network can be encoded as the following LPAD:*

alarm(t) :- burglary(t),earthquake(t).
alarm(t):0.8 ∨ alarm(f):0.2 :- burglary(t),earthquake(f).
alarm(t):0.8 ∨ alarm(f):0.2 :- burglary(f),earthquake(t).
alarm(t):0.1 ∨ alarm(f):0.9 :- burglary(f),earthquake(f).

burglary(t):0.1 ∨ burglary(f):0.9.
earthquake(t):0.2 ∨ earthquake(f):0.8.



burg	t	f
	0.1	0.9

earthq	t	f
	0.2	0.8

alarm	t	f
b=t,e=t	1.0	0.0
b=t,e=f	0.8	0.2
b=f,e=t	0.8	0.2
b=f,e=f	0.1	0.9

Figure 1.3 Bayesian network

In general, given a Bayesian network $B = \{P(X_i|\Pi_i)|i = 1,\ldots,n\}$, we obtain a ground LPAD, $\alpha(B)$, as follows. For each variable X_i and value v in the domain $D_i = \{v_{i1},\ldots,v_{im}\}$ of X_i we have one atom X_i^v in the Herbrand base of $\alpha(B)$. For each row $P(X_i|X_{i1} = v_1,\ldots,X_{il} = v_l)$ of the CPT for X_i with probabilities $p_1,\ldots,p_{im}$ for the values of X_i, we have an LPAD clause in $\alpha(B)$:

$$X_i^{v_{i1}} : p_1 \vee \ldots \vee X_i^{v_{im_i}} : p_m \text{ :- } X_{i1}^{v_1},\ldots,X_{il}^{v_l}.$$

Theorem 4. *Bayesian network B and LPAD $\alpha(B)$ define the same probability distribution.*

Proof. There is an immediate translation from a Bayesian network B to a ground BLP B'. Theorem 4 of [Vennekens and Verbaeten 2003] states the equivalence of the semantics of a ground BLP B' and a ground LPAD $\gamma(B')$ obtained from B' with a translation γ. This translation is in close correspondence with α so the theorem is proved. □

Extending Theorem 4, acyclic Datalog LPADs under the distribution semantics can be translated to Bayesian networks. To do so, first the grounding of the program must be generated. Consider an LPAD T and let $ground(T)$ be its grounding. For each atom A in the Herbrand base $\mathcal{H}_T$ of T, the network contains a binary variable A. For each clause C_i in $ground(T)$

$$H_1 : p_1 \vee \ldots \vee H_n : p_n \text{ :- } B_1,\ldots B_m,\neg C_1,\ldots,\neg C_l$$

the network contains a variable CH_i with H_1, ..., H_n and *null* as values. CH_i has

$$B_1, \ldots, B_m, C_1, \ldots, C_l$$

as parents. The CPT of CH_i is

	...	$B_1 = 1, \ldots, B_m = 1, C_1 = 0, \ldots, C_l = 0$	...
$CH_i = H_1$	0.0	p_1	0.0
...			
$CH_n = H_n$	0.0	p_n	0.0
$CH_i = null$	1.0	$1 - \sum_{i=1}^{n} p_i$	1.0

Basically, if the body assumes a false value, then CH_i assumes value *null* with certainty, otherwise the probability is distributed over atoms in the head of C_i according to the annotations.

Each variable A corresponding to atom A has as parents all the variables CH_i of clauses C_i that have A in the head. The CPT for A is:

	at least one parent equal to A	remaining columns
$A = 1$	1.0	0.0
$A = 0$	0.0	1.0

This table encodes a deterministic function: A assumes value 1 with certainty if at least one parent assumes value A, otherwise it assumes value 0 with certainty. Let us call $\lambda(T)$ the Bayesian network obtained with the above translation from an LPAD T. Then the following theorem holds.

Theorem 5. *Given an acyclic Datalog LPAD T, the Bayesian network $\lambda(T)$ defines the same probability distribution over the atoms of $\mathcal{H}_T$.*

Proof. The proof uses Theorem 5 of [Vennekens and Verbaeten 2003] that states the equivalence of the semantics of a ground LPAD T and a ground BLP $\beta(T)$ obtained from T with a translation β in close correspondence with λ. Since there is an immediate translation from a ground BLP to a Bayesian network, the theorem is proved. □

Together, Theorems 4 and 5 show the equivalence of the distribution semantics with that of Bayesian networks for the special case of acyclic probabilistic Datalog programs. As discussed in previous sections, however, the distribution semantics is defined for larger classes of programs, indicating its generality.

1.6 Inferencing in Probabilistic Logic Programs

So far, we have focused mainly on definitions and expressiveness of the distribution semantics, and this presentation has had a somewhat model-theoretic flavor. This section focuses primarily on the main inference task for probabilistic logic programs: that of query evaluation. In its simplest form, query evaluation means determining the probability of a ground query Q when no evidence is given.

Section 1.6.1 discusses the computational complexity of query evaluation which, perhaps not surprisingly, is high. Current techniques for computing the distribution semantics for stratified programs are discussed in Section 1.6.2. Because of the high computational complexity, these general techniques are not always scalable. Section 1.6.3 discusses a restriction of the distribution semantics, pioneered by the PRISM system, for which query evaluation is tractable. Another approach is to only approximate the point intervals of the distribution semantics, as discussed in Section 1.6.4. Section 1.6.4 also briefly discusses other inferencing tasks, such as computing the Viterbi probability for a query.

1.6.1 The Complexity of Query Evaluation

To understand the complexity of query evaluation for PLPs, let Q be a ground query to a probabilistic logic program T. A simple approach might be to somehow save the probabilistic choices made for each proof of Q. For instance, each time a probabilistic atom was encountered as a subgoal, the corresponding atomic choice (C, θ, i) would be added to a data structure. As a result each proof of Q would be associated with an explanation E_j, and when all n proofs of Q had been exhausted the set of explanations $\mathcal{E} = \cup_{j \leq n} E_j$ would cover Q. While this approach was sketched for top-down evaluations, a similar approach could be constructed in a bottom-up manner.

If all E_j were known to be mutually exclusive, the probability of Q $(= P(\mathcal{E}))$ could be computed simply by computing the probability of each explanation and summing them up; but this is not generally the case. Usually, explanations are not pairwise exclusive, requiring a technique such as the principle of inclusion-exclusion to be used (cf. e.g., [Rauzy et al. 2003]):

$$\begin{aligned} P(\mathcal{E}) = \quad & \textstyle\sum_{1 \leq i \leq n} P(E_i) - \sum_{1 \leq i < j \leq n} P(E_i, E_j) \\ & + \textstyle\sum_{1 \leq i < j < k \leq n} P(E_i, E_j, E_k) - \ldots + (-1)^{n+1} P(E_i, \ldots, E_n) \end{aligned} \tag{1.7}$$

Unfortunately, use of the inclusion-exclusion algorithm is exponential in n. Is there a better way? $\mathcal{E}$ can also be viewed as a propositional formula, $formula(\mathcal{E})$, in disjunctive normal form. The difficulty of determining the number of solutions to a propositional formula such as $formula(\mathcal{E})$ is the canonical #*P*-complete problem, and computing the probability of $\mathcal{E}$ is at least as difficult as computing the number of solutions of $formula(\mathcal{E})$. It can easily be shown that computing the probability of $\mathcal{E}$ also is in #*P* so that it is a #*P*-complete problem. For practical purposes, computing the probability of $\mathcal{E}$ can be thought of as equivalent to a *FPspace* complete problem (where an *FPspace* problem outputs a value, unlike a *Pspace* problem)[8].

[8] It is easy to see that counting solutions to a #*P*-complete problem can be done in polynomial space. By Toda's Theorem, every problem in *FPspace* is reducible in polynomial time to a problem in #*P* (cf. [Papadimitriou 1994]).

1.6.2 Exact Query Evaluation for Unrestricted Programs

At this point, there have been two classes of approaches to exact query evaluation for programs in which the use of the distribution semantics is unrestricted (although the programs themselves may be restricted): transformational and direct approaches. As mentioned above, we focus on probabilistic queries without evidence.

1.6.2.1 Transformational Approaches

Given the relationship between an acyclic Datalog probabilistic logic program T and a Bayesian Network as stated in Theorem 5 of Section 1.5, one approach is to transform T into a Bayesian network, use Bayesian Network inference algorithms to evaluate the query, and then translate back the results. Given the large amount of work on efficiently evaluating Bayesian networks (cf. [Koller and Friedman 2009]), such an approach could lead to efficient evaluations.

This approach was used in CVE inferencing [Meert et al. 2008, 2009], which evaluated CP-logic [Vennekens et al. 2009], a formalism closely related to LPADs. Some of the factors of the Bayesian network that results from the translation contain redundant information since they have many identical columns. To reduce the size of the generated network, this situation, called *contextual independence*, can be exploited during inference using a special technique called contextual variable elimination [Poole and Zhang 2003]. CVE applies this technique to compute the probability of queries to CP-logic programs.

An alternative approach is taken by a very recent implementation of the ProbLog system (called ProbLog2 to distinguish it from previous implementations, [Fierens et al. 2015]), which converts a program, queries and evidence (if any) to a weighted Boolean formula (cf. [Chavira and Darwiche 2008]). Once transformed, the program can be evaluated by an external weighted model counting or max-SAT solver.

1.6.2.2 Direct Approaches Based on Explanation

A more direct approach is to find a set of explanations that is covering for a query Q and then to make the explanations pairwise incompatible. Explanations can be made pairwise incompatible in a number of ways. The pD engine [Fuhr 2000] uses inclusion-exclusion (Equation 1.7) directly. The Ailog2 system for Independent Choice Logic [Poole 2000]), iteratively applies the Splitting Algorithm (Figure 1.2). More commonly however, Binary Decision Diagrams (BDDs) [Bryant 1992] are used to ensure pairwise incompatibility. This approach was first used in the ProbLog system [De Raedt et al. 2007], and later adopted by several other systems including `cplint` [Riguzzi 2007, 2009] and PITA [Riguzzi and Swift 2013]

The BDD data structure was designed to efficiently store Boolean functions (i.e., formulas), which makes it a natural candidate to store explanations. A BDD is a directed acyclic graph, with a root node representing the start of the function, and with terminal nodes 0 (false) and

1 (true). An interior node, n_i, sometimes called a decision node, represents a variable v_i in the Boolean function. Each such n_i has a 0-child representing the next node whose truth value will be examined if v_i is false, and a 1-node representing the next node whose truth value will be examined if v_i is true. Accordingly, each path from root to terminal node in a BDD represents a (partial or total) truth assignment to the variables leading to the truth of falsity of the formula. What gives a BDD its power are the following operations.

- *Ordering*: all paths through the BDD traverse variables in the same order. This ensures that each variable is traversed at most once on a given path.
- *Reduction*: within a BDD isomorphic subgraphs are merged, and any node whose two children root isomorphic subgraphs (or the same subgraph) is considered redundant and removed from the BDD. These operations ensure that once enough variables have been traversed to determine the value of the Boolean function, no other variables will be traversed (or need to be stored).

Although performing these operations when building a BDD can be expensive, the resulting BDD has the property that any two distinct paths differ in the truth value of at least one variable, so that BDDs are an efficient way to store and manipulate pairwise incompatible explanations as described in Section 1.3.3.1.

To explain the details, consider an application to ProbLog, where in each probabilistic fact, either an atom or *null* may be chosen. Let (C,θ,i) be an atomic choice for the selection of the ground probabilistic fact: $(C,\theta,1)$ means that $C\theta$ was chosen, and $(C,\theta,2)$ means that null was chosen. If we consider these atomic choices as Boolean random variables, then a set of explanations is simply a DNF formula, and storing this formula in a BDD will ensure pairwise incompatibility of the explanations in the set. Recall that if K is a pairwise incompatible set of explanations that is covering for a probabilistic query Q, then the probability of Q is given by

$$P(Q) = \sum_{\kappa \in K} \prod_{(C,\theta,i) \in \kappa} (P((C,\theta,i)).$$

Accordingly, once K is stored in a BDD, a simple traversal of the BDD suffices to compute the probability of Q as shown in Figure 1.4.

Example 20. *Returning to the sneezing example of Section 1.2, a set of covering explanations for sneezing(david) is* $K = \{\kappa_1, \kappa_2\}$, *where* $\kappa_1 = \{(C_1, \{X/david\}, 1)\}$ *and* $\kappa_2 = \{(C_2, \{X/david\}, 1)\}$. *A BDD representing K is shown in Figure 1.5: while K is not pairwise incompatible, note that the ordering and reduction operations used in constructing the BDD result in the fact that all paths through the BDD represent pairwise incompatible explanations. Using this BDD, the probability of* sneezing(david) *can be calculated by the simple algorithm of Figure 1.4.*

node_prob(BDD node n)

if n is the 1-terminal return 1

if n is the 0-terminal return 0

let t_{child} be the 1-child of n and let f_{child} be the 0-child of n and

return $P((C,\theta,1)) \times node\_prob(t_{child}) + (1 - P((C,\theta,1))) \times node\_prob(f_{child})$

Figure 1.4 Determining the probability of a node in a BDD used to store a covering set of explanations [De Raedt et al. 2007]

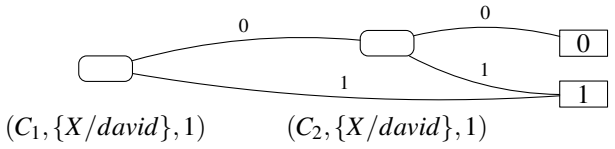


Figure 1.5 A BDD representing a pairwise incompatible set of explanations for *sneezing(david)*

Implementations of BDD-Based Explanation Approaches The ProbLog system [Kimmig et al. 2011], implemented using YAP Prolog [Costa et al. 2012], has a two-phase approach to computing probabilities for ProbLog programs. A source-code transformation is made of a ProbLog program so that during the SLD-proof phase each atomic choice is added to a running list representing the explanation of the proof; when the proof is completed the explanation is stored in a trie of the style used for tabling in YAP and XSB. Once all proofs have been completed the trie is traversed so that a BDD can be efficiently created using an external BDD package[9].

The `cplint` system [Riguzzi 2007] is also implemented using YAP Prolog and an external BDD package, but implements LPADs. To directly implement LPADs, two extensions must be made. First, the BDD interface must be modified to support ground atomic choices that allow more than two outcomes; second default negation is supported via SLDNF. The PITA system [Riguzzi and Swift 2013], is based on XSB [Swift and Warren 2012] and uses tabling extended with answer subsumption in order to combine different explanations. As each new explanation is derived for a given subgoal G, it is added to the current BDD for G. When a tabling technique termed call subsumption is also used, PITA can be shown to theoretically terminate on *any* finitely well-defined LPAD that is stratified and for which all worlds have finite models.

[9] The Cudd package (`http://vlsi.colorado.edu/~fabio/CUDD/node7.html`) is used for ProbLog as well as for `cplint` and for PITA.

Papers about CVE, BDD-based ProbLog, cplint, and PITA have compared the systems on certain probabilistic programs. Not surprisingly, source code transformations outperform meta-interpretations. The current generation of BDD-based systems usually — but not always — outperforms the transformation-based CVE, while recent experiments in [Fierens et al. 2015] indicate that translation of probabilistic logic programs into weighted Boolean formulas outperforms the use of BDDs on certain programs. ProbLog and PITA, which are more closely related, show conflicting experimental results. Timings based on PITA have shown that for traversals of probabilistic networks, the vast majority of time is spent in BDD manipulation. Based on the current set of experiments and benchmarks, there is no clear evidence about whether it is more efficient to construct a BDD during the course of evaluation as with PITA or to wait until the end as with ProbLog. In short, much more implementational and experimental work is needed to determine the best way to evaluate queries for unrestricted probabilistic logic programs.

Overall, implementation of probabilistic reasoning for the ASP-based P-Log has received less attention, although [Gelfond et al. 2006] describe a prototype implementation; while [Anh et al. 2008] describe an approach that grounds a P-Log program using XSB and then sends it to an ASP solver.

1.6.3 Exact Query Evaluation for Restricted Programs

As an alternative to inference for unrestricted programs, an implementation may be restricted to programs for which the probability of queries is easy to compute. In particular, an implementation may assume axioms of *exclusion* and *independence*. This approach has been followed in an early implementation of PHA [Poole 1993b], and in a module of the PITA system [Riguzzi and Swift 2011]; however this approach has been most thoroughly developed in the PRISM system [Sato et al. 2010], implemented using B-Prolog [Zhou 2012][10].

In order to define when an LPAD T satisfies the assumption of exclusion, consider its grounding $ground(T)$. If there is no world w of T such that the bodies of a pair of clauses of $ground(T)$ sharing an atom in the head are both true, then T satisfies the *assumption of exclusion*. In this case, a covering set of explanations will always be pairwise incompatible and the probability of a query can be computed by summing the probability of the individual explanations. As an example, the program

q :- a. *a:0.2.*
q :- a,b. *b:0.4.*

violates the exclusiveness assumption as the two clauses for the ground atom q have non-exclusive bodies.

[10] The assumptions of exclusion and independence are made in the PRISM system, but not in the PRISM language [Sato and Kameya 1997].

An LPAD T satisfies the *assumption of independence* when, for each clause C of $ground(T)$, no pair of literals in the body of C depends on a common subgoal. The assumption of independence allows the probability of a body to be computed as the product of the probabilities of its literals. As an example, the program

```
q :- a,b.
a :- c.          b :- c.          c:0.2.
```

doesn't satisfy the independence assumption because a and b both depend on c. In the distribution semantics the probability of q is 0.2, while if we multiply the probabilities of a and b in the body of the clause for q we get 0.04.

To get an idea about how restrictive these assumptions are in practice, consider the examples introduced so far in this chapter. The sneezing examples (Examples 1–8) violate the exclusion assumption; the path example (Example 10) violates both independence and exclusion; the barber example (Example 16) violates independence. However the examples about Mendelian inheritance (Example 9), Hidden Markov Models (Example 11) and alarm (Example 1.5.2) satisfy both assumptions. In terms of practical applications, programs with the independence and exclusion assumptions have been used for parameter learning [Sato and Kameya 2001], and for numerous forms of generative modeling [Sato and Kameya 2008].

PRISM implements queries to probabilistic logic programs with the independence and exclusion assumptions by using tabling to collect a set of explanations that has any duplicates filtered out. Probabilities are then collected directly from this set. PITA also uses tabling, but with the addition of answer subsumption to combine probabilities of different explanations as query evaluation progresses. In either case, computation is much faster than if the independence and exclusion assumptions do not hold. In particular, the learning algorithm of PRISM in the case of HMMs achieves the same complexity of the Baum-Welch algorithm that is specific of HMMs [Sato and Kameya 2001].

Additionally, projecting out superfluous (non-discriminating) arguments from subgoals using the technique of [Christiansen and Gallagher 2009] can lead to significant speed improvement for Hidden Markov Model examples. Finally, [Riguzzi 2012] presents approaches for efficient evaluation of probabilistic logic programs that do not use the full independence and exclusion assumptions.

1.6.4 Approximation and Other Inferencing Tasks

For programs that violate the independence or exclusion assumptions, and for which exact inference may be too expensive, approximate inference may be considered. Recall from Equation 1.7 that, using the inclusion-exclusion principle, computing probability is exponential in the number of explanations. Accordingly ProbLog [Kimmig et al. 2011] supports an optimization that retains only the k most likely explanations, thus reducing the cost of building a BDD to make the explanations pairwise incompatible. ProbLog also offers an approach similar to

iterative deepening, where lower and upper bounds on the probability are iteratively computed and inference terminates when their difference is below a given threshold. Both of these approximations are sound only for definite programs; if negation is used, sound approximation requires a three-valued semantics (Section 1.3.4.1) to distinguish the known probabilities of a query and negation from the range that is still unknown due to approximation.

Monte Carlo simulations are also used by various systems. Monte Carlo in PRISM performs Bayesian inference (updating a prior probability distribution in the light of evidence) by updating a Metropolis-Hastings algorithm for Probabilistic Context Free Grammars [Sato 2011]. ProbLog and PITA perform plain Monte Carlo by sampling the worlds and counting the fraction where the query is true, exploiting tabling to save computation.

Lastly, while this section has focused on evaluation of ground queries when there is no additional supporting evidence, this is by no means the only inference problem that has been studied. ProbLog2 [Fierens et al. 2015] evaluates queries with and without supporting evidence. PRISM supports Maximum A Posteriori (or Most Probable Explanation) inference, which finds the most likely state of a set of query atoms given some evidence. In Hidden Markov Models, this inference reduces to finding the most likely sequence of the state variables also called the Viterbi path (also supported by PITA). Again, thanks to the exclusion and independence assumptions, the complexity of finding the Viterbi path in HMMs with PRISM is the same of the Viterbi algorithm that is specific to HMMs. Finally, recent work seeks to perform inference in a lifted way, i.e., by avoiding grounding the model as much as possible. This technique can lead to exponential savings in some cases [Bellodi et al. 2014, Van den Broeck et al. 2014].

1.7 Discussion

This chapter has described the distribution semantics for logic programs, starting with stratified Datalog programs, then showing how the semantics can be extended to programs that include function symbols and non-stratified negation (Section 1.3). Various PLP languages have been described and their inter-translatability has been discussed (Section 1.2). The relationship of PLPs and Bayesian networks has also been shown (Section 1.5). Finally, the intractable problem of inferencing with the distribution semantics was discussed in Section 1.6 along with implementations that either directly address the full distribution semantics; make simplifying restrictions about the types of programs for which they provide inference; or perform heuristic approximations.

We believe that this material provides necessary background for much of the current research into PLP. However as noted, our focus on the distribution semantics leaves out many interesting and important languages and systems (a few of which were summarized in Section 1.4). In addition, we have not covered the important problem of using these languages for machine learning. Indeed, the support for machine learning has been an important motivation

for PLPs since the very first proposals and nowadays a variety of systems are available for learning either the parameters or the structure of programs under the distribution semantics.

To mention a very few such systems, PRISM [Sato and Kameya 2001], LeProbLog [Gutmann et al. 2008], LFI-ProbLog [Gutmann et al. 2011b], EMBLEM [Bellodi and Riguzzi 2013] and ProbLog2 [Fierens et al. 2015] learn the parameters either by using an EM algorithm or by gradient descent. SEM-CP-logic [Meert et al. 2008], SLIPCASE [Bellodi and Riguzzi 2012] and SLIPCOVER [Bellodi and Riguzzi 2015] learn both the structure and the parameters by performing a search in the space of possible programs and using parameter learning as a subroutine.

All these systems have been successfully applied to a variety of domains, including biology, medicine, link prediction and text classification. The results obtained show that these systems are competitive with systems at the state of the art of statistical relational learning such as Alchemy [Richardson and Domingos 2006] and others.

Acknowledgments

This work was supported by the "National Group of Computing Science (GNCS-INDAM)".

Bibliography

M. Alviano, W. Faber, and N. Leone. 2010. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10(4-6): 497–512.

H. Anh, C. Ramli, and C. Damásio. 2008. An implementation of extended P-Log using XASP. In *Proc. 24th Int. Conf. Logic Programming*, pp. 739–743.

K. R. Apt and M. Bezem. 1991. Acyclic programs. *New Generation Computing*, 9(3/4): 335–364.

F. Bacchus. 1993. Using first-order probability logic for the construction of bayesian networks. In *Proc. 9th Annual Conf. on Uncertainty in Artificial Intelligence*, pp. 219–226.

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

S. Baselice and P. Bonatti. 2010. A decidable subclass of finitary programs. *Theory and Practice of Logic Programming*, 10(4-6): 481–496.

E. Bellodi and F. Riguzzi. 2012. Learning the structure of probabilistic logic programs. In *Proc. 21st Int. Conf. on Inductive Logic Programming*, volume 7207 of *LNCS*, pp. 61–75. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31951-8_10.

E. Bellodi and F. Riguzzi. 2013. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2): 343–363.

E. Bellodi and F. Riguzzi. 2015. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(2): 169–212. doi:10.1017/S1471068413000689.

E. Bellodi, E. Lamma, F. Riguzzi, V. Santos Costa, and R. Zese. 2014. Lifted variable elimination for probabilistic logic programming. *Theory and Practice of Logic Programming*, 14(Special issue 4-5 - ICLP 2014): 681–695. doi:10.1017/S1471068414000283.

H. Blockeel. 2004. Probabilistic logical models for Mendel's experiments: An exercise. In *Proc. 14th Int. Conf. on Inductive Logic Programming*. Work in Progress Track.

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

F. Calimeri, S. Cozza, G. Ianni, and N. Leone. 2011. Finitely recursive programs: Decidability and bottom-up computation. *AI Communication*, 24(4): 311–334.

M. Chavira and A. Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7): 772—-799.

Y. Chow and H. Teicher. 2012. *Probability Theory: Independence, Interchangeability, Martingales*. Springer Texts in Statistics. Springer New York. ISBN 9781461219507.

H. Christiansen and J. Gallagher. 2009. Non-discriminating arguments and their uses. In *Proc. 25th Int. Conf. Logic Programming*, pp. 55–69.

V. S. Costa, D. Page, M. Qazi, and J. Cussens. 2003. CLP(BN): Constraint logic programming for probabilistic knowledge. In *Proc. 19th Conf. on Uncertainty in Artificial Intelligence*, pp. 517–524.

V. S. Costa, L. Damas, and R. Rocha. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2): 5–34.

J. Cussens. 2001. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3): 245–271. doi:10.1023/A:1010924021315.

E. Dantsin. 1991. Probabilistic logic programs and their semantics. In *Proc. 1st and 2nd Russian Conf. on Logic Programming*, volume 592 of *LNCS*, pp. 152–164. Springer.

L. De Raedt and A. Kimmig. 2015. Probabilistic (Logic) Programming Concepts. *Machine Learning*, 100(1): 5–47.

L. De Raedt, A. Kimmig, and H. Toivonen. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proc. 20th Int. Joint Conf. on AI*, pp. 2462–2467.

L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. 2008. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS*2008 Workshop on Probabilistic Programming*.

A. Dekhtyar and V. Subrahmanian. 2000. Hybrid probabilistic programs. *J. Logic Programming*, 43(2): 187–250.

P. Dung. 1991. Negation as hypothesis: An abductive foundation for logic programming. In *Proc. 8th Int. Conf. Logic Programming*, pp. 1–17.

D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3): 358–401.

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

M. Gelfond and V. Lifschitz. 1988. The stable model semantics for logic programming. In *Proc. 5th Int. Conf. Logic Programming*, pp. 1070–1080.

M. Gelfond and N. Rushton. 2010. Causal and probabilistic reasoning in p-log: Heuristics, probabilities and causality. In R. Dechter, H. Geffner, and J. Halpern, eds., *A Tribute to Judea Pearl*, pp. 337–359. College Publications.

M. Gelfond, N. R. N, and W. Zhu. 2006. Combining logical and probabilistic reasoning. In *Proceedings of AAAI 06 Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representationand Question Answering*, pp. 50—-55.

A. Gorlin, C. R. Ramakrishnan, and S. A. Smolka. 2012. Model checking with probabilistic tabled logic programming. *Theory and Practice of Logic Programmimg*, 12(4-5): 681–700.

S. Greco, C. Molinaro, and I. Trubitsyna. 2013. Bounded programs: A new decidable class of logic programs with function symbols. In *Proc. 23rd Int. Joint Conf. on AI*, pp. 926–932.

B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. 2008. Parameter learning in probabilistic databases: A least squares approach. In *Proc. European Conf. on Machine Learning and Knowledge Discovery in Databases*, pp. 473–488.

B. Gutmann, M. Jaeger, and L. De Raedt. 2011a. Extending problog with continuous distributions. In *Proc. 20th Int. Conf. on Inductive Logic Programming*, volume 6489 of *LNCS*, pp. 76–91. Springer.

B. Gutmann, I. Thon, and L. D. Raedt. 2011b. Learning the parameters of probabilistic logic programs from interpretations. In *Proc. European Conf. on Machine Learning and Knowledge Discovery in Databases*, pp. 581–596.

J. H. Halpern. 2003. *Reasoning About Uncertainty*. MIT Press.

M. Islam, C. R. Ramakrishnan, and I. V. Ramrkrishnan. 2012. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12(4-5): 505–523.

K. Kersting and L. D. Raedt. 2001. Towards combining inductive logic programming with Bayesian networks. In *Proc. 11th Int. Conf. on Inductive Logic Programming*, pp. 118–131.

A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3): 235–262.

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

A. N. Kolmogorov. 1950. *Foundations of the Theory of Probability*. Chelsea Publishing Company, New York.

H. Kyburg and C. Teng. 2001. *Uncertain Inference*. Cambridge University Press.

W. Meert, J. Struyf, and H. Blockeel. 2008. Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamanta Informaticae*, 89(1): 131–160.

W. Meert, J. Struyf, and H. Blockeel. 2009. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *Proc. 19th Int. Conf. on Inductive Logic Programming*.

S. Muggleton. 1996. Stochastic logic programs. In *Advances in inductive logic programming*, pp. 254–264. IOS Press.

R. Ng and V. S. Subrahmanian. 1992. Probabilistic logic programming. *Information and Computation*, 101(2): 150–201.

N. J. Nilsson. 1986. Probabilistic logic. *Artificial Intelligence*, 28(1): 71–87.

C. Papadimitriou. 1994. *Computational Complexity*. Addison-Wesley.

J. Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

D. Poole. 1993a. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1).

D. Poole. 1993b. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3): 377–400.

D. Poole. 1997. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2): 7–56.

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

D. Poole and N. Zhang. 2003. Exploiting contextual independence in probabilistic inference. *J. Artificial Intel. Res.*, 18: 266–313.

T. Przymusinski. 1989. Every logic program has a natural stratification and an iterated least fixed point model. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 11–21. ACM Press.

A. Rauzy, E. Châtelet, Y. Dutuit, and C. Bérenguer. January 2003. A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety*, 79(1): 33–42.

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

F. Riguzzi. 2007. A top-down interpreter for LPAD and CP-logic. In *Proc. 10th Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNAI*, pp. 109–120. Springer.

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

F. Riguzzi. 2012. Optimizing inference for probabilistic logic programs exploiting independence and exclusiveness. In *Proc. 27th Italian Conf. on Computational Logic*.

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

F. Riguzzi and T. Swift. March 2013. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2): 279–302. doi:10.1017/S1471068411000664.

B. Russell. 1967. Mathematical logic as based on the theory of types. In J. van Heikenoort, ed., *From Frege to Godel*, pp. 150–182. Harvard Univ. Press.

T. Sato. 1995. A statistical learning method for logic programs with distribution semantics. In *Proc. 12th Int. Conf. Logic Programming*, pp. 715–729.

T. Sato. 2011. A general MCMC method for Bayesian inference in logic-based probabilistic modeling. In *Proc. 22nd Int. Joint Conf. on AI*.

T. Sato and Y. Kameya. 1997. PRISM: A language for symbolic-statistical modeling. In *Proc. 15th Int. Joint Conf. on AI*, pp. 1330–1339.

T. Sato and Y. Kameya. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artificial Intel. Res.*, 15: 391–454.

T. Sato and Y. Kameya. 2008. New advances in logic-based probabilistic modeling by PRISM. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, eds., *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *LNCS*, pp. 118–155. Springer-Verlag. doi:10.1007/978-3-540-78652-8_5.

T. Sato and P. Meyer. 2012. Tabling for infinite probability computation. In *Proc. 28th Int. Conf. Logic Programming*, volume 17 of *LIPIcs*, pp. 348–358.

T. Sato, Y. Kameya, and N.-F. Zhou. 2005. Generative modeling with failure in PRISM. In *Proc. 19th Int. Joint Conf. on AI*, pp. 847—-852.

T. Sato, N.-F. Zhou, Y. Kameya, and Y. Izumi, 2010. PRISM User's Manual (Version 2.0). http://sato-www.cs.titech.ac.jp/prism/download/prism20.pdf.

P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. 2006. Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCS*, pp. 35–49. Springer.

T. Swift and D. S. Warren. 2012. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming*, 12(1-2): 157–187.

M. Truszczynski. 2018. An introduction to the stable and the well-founded semantics of logic programs. In M. Kifer and Y. A. Liu, eds., *Declarative Logic Programming: Theory, Systems, and Applications*. MCP/ACM.

G. Van den Broeck, W. Meert, and A. Darwiche. 2014. Skolemization for weighted first-order model counting. In *Proc. 17th Int. Conf. Principles of Knowledge Representation and Reasoning*.

A. Van Gelder, K. A. Ross, and J. S. Schlipf. 1991. The well-founded semantics for general logic programs. *J. ACM*, 38(3): 620–650.

J. Vennekens and S. Verbaeten. 2003. Logic programs with annotated disjunctions. Technical Report CW386, KU Leuven.

J. Vennekens, S. Verbaeten, and M. Bruynooghe. 2004. Logic programs with annotated disjunctions. In *Proc. 20th Int. Conf. Logic Programming*, pp. 195–209.

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

M. P. Wellman, J. S. Breese, and R. P. Goldman. 1992. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(01): 35–53.

N. Zhou. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2): 189–218.