Logic programming and uncertainty

Chitta Baral

Faculty of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-8809
chitta@asu.edu

Abstract. In recent years Logic programming based languages and features–such as rules and non-monotonic constructs–have become important in various knowledge representation paradigms. While the early logic programming languages, such as Horn logic programs and Prolog did not focus on expressing and reasoning with uncertainty, in recent years logic programming languages have been developed that can express both logical and quantitative uncertainty. In this paper we give an overview of such languages and the kind of uncertainty they can express and reason with. Among those, we slightly elaborate on the language P-log that not only accommodates probabilistic reasoning, but also respects causality and distinguishes observational and action updates.

1 Introduction

Uncertainty is commonly defined in dictionaries [1] as the state or condition of being uncertain. The adjective, uncertain, whose origin goes back to the 14th century, is ascribed the meanings, "not accurately known", "not sure" and "not precisely determined". These meanings indirectly refer to a reasoner who does not accurately know, or is not sure, or cannot determine something precisely. In the recent literature uncertainty is classified in various ways. In one taxonomy [38], it is classified to finer notions such as subjective uncertainty, objective uncertainty, epistemic uncertainty, and ontological uncertainty. In another taxonomy, uncertainty is classified based on the approach used to measure it. For example, probabilistic uncertainty, is measured using probabilities, and in that case, various possible worlds have probabilities associated with them.

Although the initial logic programming formulations did not focus on uncertainty, the current logic programming languages accommodate various kinds of uncertainty. In this overview paper we briefly discuss some of the kinds of uncertainty that can be expressed using the logic programming languages and their implications.

The early logic programming formulations are the language Prolog and Horn logic programs [13,23]. A Horn logic program, also referred to as a definite program is a collection of rules of the form: $a_0 \leftarrow a_1, \ldots, a_n.$ with $n \geq 0$ and where $a_0, \ldots, a_n$ are atoms in the sense of first order logic. The semantics of such programs can be defined using the notion of a least model or through the least fixpoint of a meaning accumulating operator [13,23].

For example, the least model of the program:

$a \leftarrow b, c.$
$d \leftarrow e.$
$b \leftarrow .$
$c \leftarrow .$

is $\{b, c, a\}$ and based on the semantics defined using the least model one can conclude that the program entails $b, c, a, \neg d$ and $\neg e$. The entailment of $\neg d$ and $\neg e$ is based on the closed world assumption [34] associated with the semantics of a Horn logic program. Thus there is no uncertainty associated with Horn logic programs.

Although Prolog grew out of Horn logic programs, and did not really aim to accommodate uncertainty, some Prolog programs can go into infinite loops with respect to certain queries and one may associate a kind of "uncertainty" value to that. Following are some examples of such programs.

P_1: $a \leftarrow a.$
$b \leftarrow .$

P_2: $a \leftarrow not\ a, c.$
$b \leftarrow .$

P_3: $a \leftarrow not\ b.$
$b \leftarrow not\ a.$
$p \leftarrow a.$
$p \leftarrow b.$

With respect to the Prolog programs P_1 and P_2 a Prolog query asking about a may take the interpreter to an infinite loop, and with respect to the program P_3 a Prolog query asking about a, a Prolog query asking about b and a Prolog query asking about p could each take the interpreter to an infinite loop.

In the early days of logic programming, such programs were considered "bad" and writing such programs was "bad programming." However, down the road, there was a movement to develop logic programming languages with clean declarative semantics, and Prolog with its non-declarative constructs was thought more as a programming language with some logical features and was not considered a declarative logic programming language. With the changed focus on clean declarative semantics, P_1, P_2 and P_3 were no longer bad programs and attempts were made to develop declarative semantics that could graciously characterize these programs as well as other syntactically correct programs. This resulted in several competing semantics and on some programs the different semantics would give different meanings. For example, for the program P_3, the stable model semantics [16] would have two different stable models $\{a, p\}$ and $\{b, p\}$ while the well-founded semantics [39] will assign the value unknown to a, b and p.

The important point to note is that unlike Horn logic programs, both stable model semantics and well-founded semantics allow characterization of some form of "uncertainty". With respect to P_3 the stable model semantics effectively encodes two possible worlds, one where a and p are true (and b is false) and another where b and p are true (and a is false). On the other hand the well-founded semantics does not delve into possible worlds; it just pronounces a, b and p to be unknown.

On a somewhat parallel track Minker and his co-authors [24] promoted the use of disjunctions in the head of logic programming rules, thus allowing explicit expression of uncertainty. An example of such a program is as follows.

P_4: $\quad a$ *or* $b \leftarrow .$
$\qquad p \leftarrow a.$
$\qquad p \leftarrow b.$

The program P_4 was characterized using its minimal models and had two minimal models $\{a, p\}$ and $\{b, p\}$. As in the case of stable models one could consider these two minimal models as two possible worlds. In both cases one can add probabilistic uncertainty by assigning probabilities to the possible models.

In the rest of the paper we give a brief overview of various logic programming languages that can express uncertainty and reason with it. We divide our overview to two parts; one where we focus on logical uncertainty without getting into numbers and another where we delve into numbers. After that we conclude and mention some future directions.

2 Logical uncertainty in logic programming

Logical uncertainty can be expressed in logic programming in various ways. In the previous section we mentioned how uncertainty can be expressed using the stable model semantics as well as using disjunctions in the head of programs. We now give the formal definition of stable models for programs that may have disjunctions in the head of rules. A logic program is then a collection of rules of the form:

$$a_0 \ or \ldots \ or \ a_k \leftarrow a_{k+1}, \ldots, a_m, \ not \ a_{m+1}, \ldots, \ not \ a_n.$$

with $k \geq 0$, $m \geq k$, $n \geq m$, and where $a_0, \ldots, a_n$ are atoms in the sense of first order logic. The semantics of such programs is defined in terms of stable models. Given such a program P, and a set of atoms S, the Gelfond-Lifschitz transformation of P with respect to S gives us a program P^S which does not have any *not* in it. This transformation is obtained in two steps as follows:

(i) All rules in P which contains *not* p in its body for some p in S are removed.

(ii) For each of the remaining rules the *not* q in the bodies of the rules are removed.

A stable model of the program P is defined as any set of atoms S such that S is a minimal model of the program P^S. An atom a is said to be true with respect to a stable model S if $a \in S$ and a negative literal $\neg a$ is said to be true with respect to a stable model S if $a \notin S$. The following examples illustrates the above definition. Consider the program

P_5: $\quad a \leftarrow not \ b.$
$\qquad b \leftarrow not \ a.$
$\qquad p$ *or* $q \leftarrow a.$
$\qquad p \leftarrow b.$

This program has three stable models $\{a, p\}$, $\{a, q\}$ and $\{b, p\}$. This is evident from noting that $P_5^{\{a,p\}}$ is the program:

$\qquad a \leftarrow .$
$\qquad p$ *or* $q \leftarrow a.$
$\qquad p \leftarrow b.$

and $\{a, p\}$ is a minimal model of $P_5^{\{a,p\}}$. Similarly, it can be shown that $\{a, q\}$ and $\{b, p\}$ are also stable models of P_5.

As we mentioned earlier, the logical uncertainty expressible using logic programs is due to both the disjunctions in the head as well as due to the possibility that even programs without disjunctions may have multiple stable models. However, in the absence of function symbols, there is a difference between the expressiveness of logic programs that allow disjunction in their head and the ones that do not. Without disjunctions the logic programs capture the class coNP, while with disjunctions they capture the class Π_2P [8].

In the absence of disjunctions, rules of the kind

P_6: $\quad a \leftarrow not\ n\_a.$
$\quad\quad n\_a \leftarrow not\ a.$

allow the enumeration of the various possibilities and rules of the form

P_7: $\quad p \leftarrow not\ p, q.$

allow elimination of stable models where certain conditions (such as q) may be true. The elimination rules can be further simplified by allowing rules with empty head. In that case the above rule can be simply written as: $\quad P_8$: $\quad \leftarrow q.$ $\quad$ When rules with empty heads, such as in P_8, are allowed, one can replace the constructs in P_6 by exclusive disjunctions [20] of the form: $\quad P_9$: $\quad a \oplus n\_a \leftarrow .$ $\quad$ to do the enumeration, and can achieve the expressiveness to capture the class coNP with such exclusive disjunctions, rules with empty heads as in P_8 and stratified negation. The paper [20] advocates this approach with the argument that many find the use of unrestricted negation to be unintuitive and complex. On the other hand use of negation is crucial in many knowledge representation tasks and while using them having not to worry whether the negation used is stratified or not makes the task simpler for humans.

2.1 Answer sets and use of classical negation

Allowing classical negation in logic programs gives rise to a different kind of uncertainty. For example the program

P_{10}: $\quad a \leftarrow b.$
$\quad\quad \neg b \leftarrow .$

has a unique answer set $\{\neg b\}$ and with respect to that answer set the truth value of a is unknown. We now give the formal definition of answer sets for programs that allows classical negation. A logic program is then a collection of rules of the form:

$$l_0\ or \ldots\ or\ l_k \leftarrow l_{k+1}, \ldots, l_m,\ not\ l_{m+1}, \ldots,\ not\ l_n.$$

with $k \geq 0$, $m \geq k$, $n \geq m$, and where $l_0, \ldots, l_n$ are literals in the sense of first order logic. The semantics of such programs is defined in terms of answer sets [17]. Given such a program P, an answer set of the program P is defined as a consistent set of literals S such that S satisfies all rules in P^S and no proper subset of S satisfies all rules of P^S, where P^S is as defined earlier. A literal l is defined to be true with respect to an answer set S if $l \in S$. With the use of classical negation one need not invent new atoms for the enumeration, as was done in P_6, and simply write:

P_{11}: $\quad a \leftarrow not\ \neg a.$
$\quad\quad \neg a \leftarrow not\ a.$

Moreover, since answer sets are required to be consistent, one need not write explicit rules of the kind: $\leftarrow a, \neg a.$ which were sometimes explicitly needed to be written when not using classical negation.

2.2 Other logic programming languages and systems for expressing logical uncertainty

Other logic programming languages that can express logical uncertainty include abductive logic programs [21] and various recent logic programming languages. Currently there are various logic programming systems that one can be use to express logical uncertainty. The most widely used are Smodels [29], DLV [11] and the Potassco suite [15].

3 Multi-valued and Quantitative uncertainty in logic programming

Beyond logical uncertainty that we discussed in the previous section where one could reason about truth, falsity and lack of knowledge using logic programming, one can classify uncertainty in logic programming in several dimensions.

- The truth values may have an associated degree of truth and falsity or we may have multi-valued truth values.
- The degree of truth or the values (in the multi-valued case) could be discrete or continuous.
- They can be associated with the whole rule or with each atom (or literal) in the rule.
- The formalism follows or does not follow axioms of probability.
- The formalism is motivated by concerns to learn rules and programs.

Examples of logic programming with more than three discrete truth values include use of bi-lattice in logic programming in [14], use of annotated logics in [6] and various fuzzy logic programming languages.

Among the various quantitative logic programming languages the recollection [37] considers Shapiro's quantitative logic programming [36] as the first "serious" paper on the topic. Shapiro assigned a mapping to each rule; the mapping being from numbers in (0,1] associated with each of the atoms in the body of the rule to a number in (0,1] to be associated with the atom in the head of the rule. He gave a model-theoretic semantics and developed a meta-interpreter. A few years later van Emden [12] considered the special case where numbers were only associated with a rule and gave a fixpoint and a sound and conditionally-complete proof theory.

A large body of work on quantitative logic programming has been done by Subrahmanian with his students and colleagues. His earliest work used the truth values [0,1] $\cup \{*\}$, where $*$ denoted inconsistency and as per the recollection [37] it was "the first work that explicitly allowed a form of negation to appear in the head." This was followed by his work on paraconsistent logic programming [6] where truth values could be from any lattice. He and his colleagues further generalized paraconsistency to generalized annotations and generalized annotated programs where complex terms could be used as annotations.

3.1 Logic Programming with probabilities

The quantitative logic programming languages mentioned earlier, even when having numbers, did not treat them as probabilities. In this section we discuss various logic programming languages that accommodate probabilities[1].

Probabilistic Logic Programming

The first probabilistic logic programming language was proposed by Ng and Subrahmanian [27]. Rules in this language were of the form:

$a_0 : [\alpha_0, \beta_0] \leftarrow a_1 : [\alpha_1, \beta_1], \ldots, a_n : [\alpha_n, \beta_n].$

with $n \geq 0$ and where $a_0, \ldots, a_n$ are atoms in the sense of first-order logic, and $[\alpha_i, \beta_i] \subseteq [0, 1]$. Intuitively, the meaning of the above rule is that if the probability of a_j being true is in the interval $[\alpha_j, \beta_j]$, for $1 \leq j \leq n$, then the probability of a_0 being true is in the interval $[\alpha_0, \beta_0]$. Ng and Subrahmanian gave a model theoretic and a fixpoint characterization of such programs and also gave a sound and complete query answering method. The semantics made the "ignorance" assumption that nothing was known about any dependencies between the events denoted by the atoms. Recently a revised semantics for this language has been given in [9].

Ng and Subrahmanian later extend the language to allow a_i's to be conjunction and disjunction of atoms and the a_is in the body were allowed to have not preceding them. In presence of not the semantics was given in a manner similar to the definition of stable models.

Dekhtyar and Subrahmanian [10] further generalized this line of work to allow explicit specification of the assumptions regarding dependencies between the events denoted by the atoms that appear in a disjunction or conjunction. Such assumptions are referred to as *probabilistic strategies* and examples of probabilistic strategies include: (i) independence, (ii) ignorance, (iii) mutual exclusion and (iv) implication. While some of the probabilistic logic programming languages assume one of these strategies and hard-code the semantics based on that, the hybrid probabilistic programs of [10] allowed one to mention the probabilistic strategies used in each conjunction or disjunction. For example, $\wedge_{ind}$ and $\vee_{ind}$ would denote the conjunction and disjunction associated with the "independence" assumption and would have the property that $Prob(e_1 \wedge_{ind} \ldots \wedge_{ind} e_n) = Prob(e_1) \times \ldots \times Prob(e_n)$. Following are examples, from [10], of rules of hybrid probabilistic programs:

$price\_drop(C) : [.4, .9] \leftarrow (ceo\_sells\_stock(C) \vee_{igd}\ ceo\_retires(C)) : [.6, 1].$
$price\_drop(C) : [.5, 1] \leftarrow (strike(C) \vee_{ind} accident(C)) : [.3, 1].$

The intuitive meaning of the first rule is that if the probability of the CEO of a company selling his/her stock or retiring is greater than 0.6 then the probability of the price dropping is between 0.4 and 0.9, and it is assumed that the relationship between the CEO retiring and selling stock is not known. The intuitive meaning of the second rule is that if the probability of a strike happening or an accident happening –which are considered to be independent–is greater than 0.3 then the probability of the price dropping is greater than 0.5.

[1] Some of these were discussed in our earlier paper [4], but the focus there was comparison with P-log.

Lukaciewicz [25] proposed the alternative of using conditional probabilities in probabilistic logic programs. In his framework clauses were of the form: $(H \mid B)[\alpha_1, \beta_1]$

where H and B are conjunctive formulas and $0 \leq \alpha_1 \leq \beta_1 \leq 1$, and a probabilistic logic program consisted of several such clauses. The intuitive meaning of the above clause is that the conditional probability of H given B is between α_1 and β_1. Given a program consisting of a set of such clauses the semantics is defined based on models where each model is a probability distribution that satisfies each of the clauses in the program.

Bayesian Logic Programming

Bayesian logic programs [22] are motivated by Bayes nets and build up on an earlier formalism of probabilistic knowledge bases [28] and add some first-order syntactic features to Bayes nets so as to make them relational. A Bayesian logic program has two parts, a logical part that looks like a logic program and a set of conditional probability tables. A rule or a clause of a Bayseian logic program is of the form: $H \mid A_1, \ldots, A_n$

where $H, A_1, \ldots, A_n$ are atoms which can take a value from a given domain associated with the atom. An example of such a clause is:

$highest\_degree(X) \mid instructorr(X)$.

Its corresponding domain could be, for example, $D_{instructor} = \{yes, no\}$, and $D_{highest\_degree} = \{phd, masters, bachelors\}$. Each such clause has an associated conditional probability table (CPT). For example, the above clause may have the following table:

instructor(X)	highest_degree(X) phd	highest_degree(X) masters	highest_degree(X) bachelors
yes	0.7	0.25	0.05
no	0.05	0.3	0.65

Acyclic Bayesian logic programs are characterized by considering their grounded versions. If the ground version has multiple rules with the same ground atom in the head then combination rules are specified to combine these rules to a single rule with a single associated conditional probability table.

Stochastic Logic Programs

Stochastic logic programs [26] are motivated from the perspective of machine learning and are generalization of stochastic grammars. Consider developing a grammar for a natural language such that the grammar is not too specific and yet is able to address ambiguity. This is common as we all know grammar rules which work in most cases but not necessarily in all cases and yet with our experience we are able to use those rules. In statistical parsing one uses a stochastic grammar where production rules have associated weight parameters that contribute to a probability distribution. Using those weight parameters one can define a probability function $Prob(w|s, p)$, where s is a sentence, w is a parse and p is the weight parameter associated with the production rules of the grammar. Given a grammar and its associated p, a new sentence s' is parsed to w' such that $Prob(w'|s', p)$ is the maximum among all possible parses of s'. The weight parameter p is learned from a given training set of example sentences and their parses. In the learning process, given examples of sets of sentences and parses $\{(s_1, w_1), \ldots, (s_n, w_n)\}$

one has to come up with the p that maximizes the probability that the s_i's in the training set are parsed to the w_i's.

Motivated by stochastic grammars and with the goal to allow inductive logic programs to have associated probability distributions, [26] generalized stochastic grammars to stochastic logic programs. In stochastic logic programs [26] a number in [0,1], referred to as a "probability label," is associated with each rule of a Horn logic program with the added conditions that the rules be range restricted and for each predicate symbol q, the probability labels for all clauses with q in the head sum to 1. Thus, a Stochastic logic program [26] P is a collection of clauses of the form

$p \ : \ a_0 \leftarrow a_1, \ldots, a_n.$

where p (referred to as the the probability label) belongs to $[0, 1]$, and $a_0, a_1, \ldots a_n$ are atoms. The probability of an atom g with respect to a stochastic logic program P is obtained by summing the probability of the various SLD-refutation of $\leftarrow g$ with respect to P, where the probability of a refutation is computed by multiplying the probability of various choices; and doing appropriate normalization. For example, if the first atom of a subgoal $\leftarrow g'$ unifies with the head of the stochastic clause $p_1 \ : \ C_1$, also with the head of the stochastic clause $p_2 \ : \ C_2$ and so on up to the head of the stochastic clause $p_m \ : \ C_m$, and the stochastic clause $p_i \ : \ C_i$ is chosen for the refutation, then the probability of this choice is $\frac{p_i}{p_1 + \cdots + p_m}$.

Modularizing probability and logic aspects: Independent Choice Logic

Earlier in Section 2 we discussed how one can express logical uncertainty using logic programming. One way to reason with probabilities in logic programming is to assign probabilities to the "possible worlds" defined by the approaches in Section 2. Such an approach is taken by Poole's Independent Choice Logic of [31,32], a refinement of his earlier work on probabilistic Horn abduction [33].

There are three components of an Independent Choice Logic of interest here: a choice space $\mathcal{C}$, a rule base $\mathcal{F}$ and a probability distribution on $\mathcal{C}$ such that $\Sigma_{X \in \mathcal{C}} Prob(X) = 1$. A Choice space $\mathcal{C}$ is a set of sets of ground atoms such that if $X_1 \in \mathcal{C}$, $X_2 \in \mathcal{C}$ and $X_1 \neq X_2$ then $X_1 \cap X_2 = \emptyset$. An element of $\mathcal{C}$ is referred to as an "alternative" and an element of an "alternative" is referred to as an "atomic choice". A rule base $\mathcal{F}$ is a logic program such that no atomic choice unifies with the head of any of its rule and it has a unique stable model. The unique stable model condition can be enforced by restrictions such as requiring the program to be an acyclic program without disjunctions. $\mathcal{C}$ and $\mathcal{F}$ together define the set of possible worlds and the probability distribution on $\mathcal{C}$ can then be used to assign probabilities to the possible worlds. These probabilities can then be used in the standard way to define probabilities of formulas and conditional probabilities.

Logic programs with distribution semantics: PRISM

The formalism of Sato [35], which he refers to as PRISM as a short form for "PRogramming In Statistical Modeling", is very similar to Independent Choice Logic. A PRISM formalism has a possibly infinite collection of ground atoms, F, the set Ω_F of all interpretations of F, and a completely additive probability measure P_F which quantifies the likelihood of the interpretations. P_F is defined on some fixed σ algebra of subsets of Ω_F.

In Sato's framework interpretations of F can be used in conjunction with a Horn logic program R, which contains no rules whose heads unify with atoms from F. Sato's logic program is a triple, $\Pi = \langle F, P_F, R \rangle$. The semantics of Π is given by a collection Ω_Π of possible worlds and the probability measure P_Π. A set M of ground atoms in the language of Π belongs to Ω_Π iff M is a minimal Herbrand model of a logic program $I_F \cup R$ for some interpretation I_F of F. The completely additive probability measure of P_Π is defined as an extension of P_F.

The emphasis of the original work by Sato and other PRISM related research is on the use of the formalism for design and investigation of efficient algorithms for statistical learning. The goal is to use the pair $DB = \langle F, R \rangle$ together with observations of atoms from the language of DB to learn a suitable probability measure P_F.

Logic programming with annotated disjunctions

In the LPAD formalism of Vennekens et al. [40] rules have choices in their head with associate probabilities. Thus an LPAD program consists of rules of the form:

$(h_1 : \alpha_1) \vee \ldots \vee (h_n : \alpha_n) \leftarrow b_1, \ldots, b_m$

where h_i's are atoms, b_is are atoms or atoms preceded by *not*, and $\alpha_i \in [0, 1]$, such that $\sum_{i=1}^{n} \alpha_i = 1$. An LPAD rule instance is of the form: $h_i \leftarrow b_1, \ldots, b_m$.

The associated probability of the above rule instance is then said to be α_i. An instance of an LPAD program P is a logic program P' obtained as follows: for each rule in P exactly one of its instance is included in P', and nothing else is in P'. The associated probability of an instance P', denoted by $\pi(P')$, of an LPAD program is the product of the associated probability of each of its rules.

An LPAD program is said to be sound if each of its instance has a 2-valued well-founded model. Given an LPAD program P, and a collection of atoms I, the probability assigned to I by P is given as follows:

$$\pi_P(I) = \sum_{P' \text{ is an instance of } P \text{ and } I \text{ is the well-founded model of } P'} \pi(P')$$

The probability of a formula ϕ assigned by an LPAD program P is then defined as:

$$\pi_P(\phi) = \sum_{\phi \text{ is satisfied by } I} \pi_P(I)$$

4 Logic Programming with probabilities, causality and generalized updates: P-log

An important design aspect of developing knowledge representation languages and representing knowledge in them is to adequately address how knowledge is going to be updated. If this is not thought through in the design and representation phase then updating a knowledge base may require major surgery. For this reason updating a knowledge base in propositional logic or first-order logic is hard. This is also one of the motivations behind the development of non-monotonic logics which have constructs that allow elaboration tolerance.

The probabilistic logic programming language P-log was developed with updates and elaboration tolerance in mind. In particular, it allows one to easily change the domain of the event variables. In most languages the possible values of a random variable get restricted with new observations. P-log with its probabilistic non-monotonicity allows the other way round too. Another important aspect of updating in P-log is that it differentiates between updating due to new observations and updating due to actions; this is especially important when expressing causal knowledge.

Elaborating on the later point, an important aspect of probabilistic uncertainty that is often glossed over is the proper representation of joint probability distributions. Since the random variables in a joint probability distribution are often not independent of each other, and since representing the joint probability distribution explicitly is exponential in the number of variables, techniques such as Bayseian networks are used. However, as pointed out by Pearl [30], such representations are not amenable to distinguish between observing the value of a variable and execution of actions that change the value of the variable. As a result prior to Pearl (and even now) most probability formalisms are not able to express action queries such as the probability that X has value a given that Y's value is made to be b. Note that this is different from the query about the probability that X has value a given that Y's value is observed to be b. To be able to address this accurately a causal model of probability is needed. P-log takes that view and is able to express both the above kind of queries and distinguishes between them.

With the above motivations we give a brief presentation on P-log[2] starting with its syntax and semantics and following up with several illustrative examples.

A P-log program consists of a declaration of the domain, a logic program without disjunctions, a set of random selection rules, a set of probability atoms, and a collection of observations and action atoms.

The declaration of the domain consists of sort declarations of the form $c = \{x_1, \dots, x_n\}$. or consists of a logic program T with a unique answer set A. In the latter case $x \in c$ iff $c(x) \in A$. The domain and range of attributes[3] are given by statements of the form: $a : c_1 \times \cdots \times c_n \to c_0$.

A random selection rule is of the form

$$[\, r\,]\ random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B. \tag{1}$$

where r is a term used to name the rule and B is a collection of extended literals of the form l or $not\ l$, where l is a literal. Statement (1) says that *if B holds, the value of $a(\bar{t})$ is selected at random from the set $\{X : p(X)\} \cap range(a)$ by experiment r, unless this value is fixed by a deliberate action.*

A probability atom is of the form: $pr_r(a(\bar{t}) = y \mid_c B) = v$. where $v \in [0, 1]$, B is a collections of extended literals, *pr* is a special symbol, r is the name of a random selection rule for $a(\bar{t})$, and $pr_r(a(\bar{t}) = y \mid_c B) = v$ says that *if the value of $a(\bar{t})$ is fixed by experiment r, and B holds, then the probability that r causes $a(\bar{t}) = y$ is v.*

[2] Our presentation is partly based on our earlier paper [4].

[3] Attributes are relational variables. In probabilistic representations, a variable such as Color can take the value from {red, green, blue, ... }. Now if we want talks about colors of cars, then color is a function from a set of cars to {red, green, blue, ... }. In that case we call “color” an attribute.

(Note that here we use 'cause' in the sense that B is an immediate or proximate cause of $a(\bar{t}) = y$, as opposed to an indirect cause.)

Observations and action atoms are of the form: $obs(l).$ $do(a(\bar{t}) = y)).$

where l is a literal. Observations are used to record the outcomes of random events, i.e., random attributes, and attributes dependent on them.

We now illustrate the above syntax using an example from [4] about certain dices being rolled. In that example, there are two dices owned by Mike and John respectively. The domain declarations are given as follows:

$dice = \{d_1, d_2\}.$
$score = \{1, 2, 3, 4, 5, 6\}.$
$person = \{mike, john\}.$
$roll : dice \to score.$
$owner : dice \to person.$
$even : dice \to Boolean.$

The logic programming part includes the following:

$owner(d_1) = mike.$
$owner(d_2) = john.$
$even(D) \leftarrow roll(D) = Y, Y \; mod \; 2 = 0.$
$\neg even(D) \leftarrow not \; even(D).$

The fact that values of attribute $roll : dice \to score$ are random is expressed by the statement

$[\, r(D) \,] \; random(roll(D))$

The dice domain may include probability atoms that convey that the die owned by John is fair, while the die owned by Mike is biased to roll 6 at a probability of .25.

Let us refer to the P-log program consisting of the above parts as T_1.

$pr(roll(D) = Y \mid_c owner(D) = john) = 1/6.$
$pr(roll(D) = 6 \mid_c owner(D) = mike) = 1/4.$
$pr(roll(D) = Y \mid_c Y \neq 6, owner(D) = mike) = 3/20.$

In this domain the observation $\{obs(roll(d_1) = 4)\}$ records the outcome of rolling dice d_1. On the other hand the statement $\{do(roll(d_1) = 4)\}$ indicates that d_1 was simply put on the table in the described position. One can have observations such as $obs(even(d_1))$ which means that it was observed that the dice d_1 had an even value. Here, even though $even(d_1)$ is not a random attribute, it is dependent on the random attribute $roll(d_1)$.

The semantics of a P-log program is given in two steps. First the various parts of a P-log specification is translated to logic programs and then the answer sets of the translated program is computed and are treated as possible worlds and probabilities are computed for them. The translation of a P-log specification Π to a logic program $\tau(\Pi)$ is as follows:

1. Translating the declarations: For every sort declaration $c = \{x_1, \ldots, x_n\}$ of Π, $\tau(\Pi)$ contains $c(x_1), \ldots, c(x_n)$. For all sorts that are defined using a logic program T in Π, $\tau(\Pi)$ contains T.
2. Translating the Logic programming part:

   (a) For each rule r in the logic programming part of Π, $\tau(\Pi)$ contains the rule obtained by replacing each occurrence of an atom $a(\bar{t}) = y$ in r by $a(\bar{t}, y)$.
   (b) For each attribute term $a(\bar{t})$, $\tau(\Pi)$ contains the rule:

$$\neg a(\bar{t}, Y_1) \leftarrow a(\bar{t}, Y_2), Y_1 \neq Y_2. \tag{2}$$

   which guarantees that in each answer set $a(\bar{t})$ has at most one value.
3. Translating the random selections:
   (a) For an attribute a, we have the rule: $intervene(a(\bar{t})) \leftarrow do(a(\bar{t}, Y))$. where, intuitively, $intervene(a(\bar{t}))$ means that the value of $a(\bar{t})$ is fixed by a deliberate action. Semantically, $a(\bar{t})$ will not be considered random in possible worlds which satisfy $intervene(a(\bar{t}))$.
   (b) Each random selection rule of the form

$$[\, r \,]\ random(a(\bar{t}) : \{Z : p(Z)\}) \leftarrow B.$$

   with $range(a) = \{y_1, \ldots, y_k\}$ is translated to the following rule:

$$a(\bar{t}, y_1) \text{ or } \ldots \text{ or } a(\bar{t}, y_k) \leftarrow B,\ \text{ not }\ intervene(a(\bar{t})) \tag{3}$$

   If the dynamic range of a in the selection rule is not equal to its static range, i.e. expression $\{Z : p(Z)\}$ is not omitted, then we also add the rule

$$\leftarrow a(\bar{t}, y),\ \text{ not }\ p(y), B,\ \text{ not }\ intervene(a(\bar{t})). \tag{4}$$

   Rule (3) selects the value of $a(\bar{t})$ from its range while rule (4) ensures that the selected value satisfies p.
4. $\tau(\Pi)$ contains actions and observations of Π.
5. For each Σ-literal l, $\tau(\Pi)$ contains the rule: $\leftarrow obs(l),\ \text{ not }\ l$.
6. For each atom $a(\bar{t}) = y$, $\tau(\Pi)$ contains the rule: $a(\bar{t}, y) \leftarrow do(a(\bar{t}, y))$.
   The last but one rule guarantees that no possible world of the program fails to satisfy observation l. The last rule makes sure the atoms that are made true by the action are indeed true.

The answer sets of the above translation are considered the possible worlds of the original P-log program. To illustrate how the above translation works, $\tau(T_1)$ of T_1 will consist of the following:
$dice(d_1).\ dice(d_2).\ score(1).\ score(2).$
$score(3).\ score(4).\ score(5).\ score(6).$
$person(mike).\ person(john).$
$owner(d_1, mike).\ owner(d_2, john).$
$even(D) \leftarrow roll(D, Y), Y\ mod\ 2 = 0.$
$\neg even(D) \leftarrow\ \text{ not }\ even(D).$
$\neg roll(D, Y_1) \leftarrow roll(D, Y_2), Y_1 \neq Y_2.$
$\neg owner(D, P_1) \leftarrow owner(D, P_2), P_1 \neq P_2.$
$\neg even(D, B_1) \leftarrow even(D, B_2), B_1 \neq B_2.$
$intervene(roll(D)) \leftarrow do(roll(D, Y)).$

$roll(D,1)$ or $\ldots$ or $roll(D,6) \leftarrow B$, not $intervene(roll(D))$.
$\leftarrow obs(roll(D,Y))$, not $roll(D,Y)$.
$\leftarrow obs(\neg roll(D,Y))$, not $\neg roll(D,Y)$.
$roll(D,Y)) \leftarrow do(roll(D,Y))$.

The variables D, P, B's, and Y's range over $dice$, $person$, $boolean$, and $score$ respectively.

Before we explain how the probabilities are assigned to the possible worlds, we mention a few conditions that the P-log programs are required to satisfy. They are:

(i) There can not be two random selection rules about the same attribute whose bodies are simultaneously satisfied by a possible world.

(ii) There can not be two probability atoms about the same attribute whose conditions can be simultaneously satisfied by a possible world.

(iii) A random selection rule can not conflict with a probability atom in such a way that probabilities are assigned outside the range given in the random selection rule.

The probabilities corresponding to each of the possible worlds are now computed in the following way:

(a) Computing an initial probability assignment P for each atom in a possible world: For a possible world W if the P-log program contains $pr_r(a(\bar{t}) = y \mid_c B) = v$ where r is the generating rule of $a(\bar{t}) = y$, W satsifies B, and W does not contain $intervene(a(\bar{t}))$, then $P(W, a(\bar{t}) = y) = v$.

(b) For any $a(\bar{t})$, the probability assignments obtained in step (a) are summed up and for the other possible values of $a(\bar{t})$ the remaining probability (i.e., 1 - the sum) is uniformly divided.

(c) The *unnormalized probability*, $\hat{\mu}_T(W)$, of a possible world W *induced by* a given P-log program T is $\hat{\mu}_T(W) = \prod_{a(\bar{t},y)\in\, W} P(W, a(\bar{t}) = y)$ where the product is taken over atoms for which $P(W, a(\bar{t}) = y)$ is defined. The above measure is then normalized to $\mu_T(W)$ so that the sum of it for all possible worlds W is 1.

Using the above measure, the probability of a formula F with respect to a program T is defined as $Prob_T(F) = \Sigma_{W \models F} \mu_T(W)$.

We now show how P-log can be used to express updates not expressible in other probabilistic logic programming languages. Lets continue with the dice rolling example. Suppose we have a domain where the dices are normally rigged to roll 1 but once in a while there may be an abnormal dice that rolls randomly. This can be expressed in P-log by:

$roll(D) = 1 \leftarrow$ *not* $abnormal(D)$
$random(roll(D)) \leftarrow abnormal(D)$

Updating such a P-log program with $obs(abnormal(d_1))$ will expand the value that $roll(d_1)$ can take.

Now let us consider an example that illustrates the difference between observational updates and action updates. Lets augment the dice domain with a new attribute $fire\_works$ which becomes true when dice d_1 rolls to 6. This can be expressed by the rule:

$fire\_works \leftarrow roll(d_1) = 6$.

Now suppose we observe fire works. This observation can be added to the P-log program as $obs(fire\_works)$, and when this observation is added to the P-log program it will eliminate the earlier possible worlds where $fire\_works$ was not true and as a result the probability that dice d_1 was rolled 6 will increase to 1. Now supposed instead of observing the fire works someone goes and starts the fire work. In that case the update to the P-log program would be $do(fire\_works = true)$. This addition will only add $fire\_works$ to all the previous possible worlds and as a result the probability that dice d_1 was rolled 6 will remain unchanged.

As suggested by the above examples, updating a P-log program basically involves adding to it. Formally, the paper [4] defines a notion of coherence of P-log programs and uses it to define updating a P-log program T by U as addition of U to T with the requirement that $T \cup U$ be coherent. The paper also shows that the traditional conditional probability $Prob(A|B)$ defined as $\frac{Prob(A \wedge B)}{Prob(B)}$ is equal to the $Prob_{T \cup obs(B)}(A)$ where $obs(B) = \{obs(l) : l \in B\}$.

Since the original work on P-log [3,4] which we covered in this section there have been several new results. This includes work on using P-log to model causality and counterfactual reasoning [5], implementation of P-log [19], an extension of P-log that allows infinite domains [18] and modular programming in P-log [7].

5 Conclusion and Future directions

In this paper we have given a personal overview of representing and reasoning about uncertainty in logic programming. We started with a review of representing logical uncertainty in logic programming and then discussed some of the multi-valued and quantitative logic programming languages. We briefly discussed some of the probabilistic logic programming languages. Finally we discussed logic programming languages that have distinct logical and probabilistic components and concluded with the language of P-log that has distinct logical and probabilistic components, that allows a rich variety of updates and makes a distinction between observational updates and action updates. Our overview borrowed many examples, definitions and explanations from the book [2] and the articles [37] and [3,4]. We refer the reader to those articles and the original papers for additional details.

Although a lot has been done, there still is a big gap between knowledge representation (KR) languages that are used by humans to encode knowledge, KR languages that are learned and KR languages used in translating natural language to a formal language. We hope these gaps will be narrowed in the future, and to that end we need to develop ways to learn theories in the various logic programming languages that can express and reason with uncertainty. For example, it remains a challenge to explore how techniques from learning Bayes nets and statistical relational learning can be adapted to learn P-log theories. P-log also needs more efficient interpreters and additional refinements in terms of explicitly expressing probabilistic strategies.

References

1. Online etymology dictionary (Jul 2011), http://dictionary.reference.com/browse/uncertain
2. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)

3. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. In: Proceedings of LPNMR7. pp. 21–33 (2004)
4. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. TPLP 9(1), 57–144 (2009)
5. Baral, C., Hunsaker, M.: Using the probabilistic logic programming language p-log for causal and counterfactual reasoning and non-naive conditioning. In: IJCAI. pp. 243–249 (2007)
6. Blair, H., Subrahmanian, V.: Paraconsistent logic programming. Theoretical Computer Science 68, 135–154 (1989)
7. Damasion, C., Moura, J.: Modularity of P-log programs. In: LPNMR. pp. 13–25 (2011)
8. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. In: Proc. of 12th annual IEEE conference on Computational Complexity. pp. 82–101 (1997)
9. Dekhtyar, A., Dekhtyar, M.: Possible worlds semantics for probabilistic logic programs. In: ICLP. pp. 137–148 (2004)
10. Dekhtyar, A., Subrahmanian, V.S.: Hybrid probabilistic programs. Journal of Logic Programming 43(3), 187–250 (2000)
11. Eiter, T., Faber, W., Gottlob, G., Koch, C., Mateis, C., Leone, N., Pfeifer, G., Scarcello, F.: The dlv system. In: Minker, J. (ed.) Pre-prints of Workshop on Logic-Based AI (2000)
12. van Emden, M.: Quantitative deduction and its fixpoint theory. The Journal of Logic Programming 3(2), 37–53 (1986)
13. van Emden, M., Kowalski, R.: The semantics of predicate logic as a programming language. Journal of the ACM. 23(4), 733–742 (1976)
14. Fitting, M., Ben-Jacob, M.: Stratified and Three-Valued Logic programming Semantics. In: Kowalski, R., Bowen, K. (eds.) Proc. 5^{th} International Conference and Symposium on Logic Programming. pp. 1054–1069. Seattle, Washington (August 15-19, 1988)
15. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Communications - Answer Set Programming archive 24(2) (2011)
16. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Logic Programming: Proc. of the Fifth Int'l Conf. and Symp. pp. 1070–1080. MIT Press (1988)
17. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Warren, D., Szeredi, P. (eds.) Logic Programming: Proc. of the Seventh Int'l Conf. pp. 579–597 (1990)
18. Gelfond, M., Rushton, N.: Causal and probabilistic reasoning in p-log (to appear in an edited book)
19. Gelfond, M., Rushton, N., Zhu, W.: Combining logical and probabilistic reasoning. In: AAAI Spring 2006 Symposium. pp. 50–55 (2006)
20. Greco, S., Molinaro, C., Trubitsyna, I., Zumpano, E.: NP-Datalog: A logic language for expressing search and optimization problems. TPLP 10(2), 125–166 (2010)
21. Kakas, A., Kowalski, R., Toni, F.: Abductive logic programming. Journal of Logic and Computation 2(6), 719–771 (1993)
22. Kersting, K., Raedt, L.D.: Bayesian logic programs. In: Cussens, J., Frisch, A. (eds.) Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming. pp. 138–155 (2000)
23. Lloyd, J.: Foundations of logic programming. Springer (1984)
24. Lobo, J., Minker, J., Rajasekar, A.: Foundations of disjunctive logic programming. The MIT Press (1992)
25. Lukasiewicz, T.: Probabilistic logic programming. In: ECAI. pp. 388–392 (1998)
26. Muggleton, S.: Stochastic logic programs. In: De Raedt, L. (ed.) Proceedings of the 5th International Workshop on Inductive Logic Programming. p. 29. Department of Computer Science, Katholieke Universiteit Leuven (1995)

27. Ng, R.T., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
28. Ngo, L., Haddawy, P.: Answering queries from context-sensitive probabilistic knowledge bases. Theoretical Computer Science 171(1–2), 147–177 (1997)
29. Niemelä, I., Simons, P.: Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In: Dix, J., Furbach, U., Nerode, A. (eds.) Proc. 4th international conference on Logic programming and non-monotonic reasoning. pp. 420–429. Springer (1997)
30. Pearl, J.: Causality. Cambridge University Press (2000)
31. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94(1-2 (special issue on economic principles of multi-agent systems)), 7–56 (1997)
32. Poole, D.: Abducing through negation as failure: Stable models within the independent choice logic. Journal of Logic Programming 44, 5–35 (2000)
33. Poole, D.: Probabilistic horn abduction and bayesian networks. Artificial Intelligence 64(1), 81–129 (1993)
34. Reiter, R.: On closed world data bases. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 119–140. Plenum Press, New York (1978)
35. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Proceedings of the 12th International Conference on Logic Programming (ICLP95). pp. 715–729 (1995)
36. Shapiro, E.: Logic programs with uncertainties: A tool for implementing expert systems. In: Proc. IJCAI (1983)
37. Subrahmanian, V.S.: Uncertainty in logic programming: some recollections. ALP Newsletter (May 2007)
38. Tannert, C., Elvers, H., Jandrig, B.: The ethics of uncertainty. in the light of possible dangers, research becomes a moral duty. EMBO Report 8(10), 892–896 (2007)
39. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)
40. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: ICLP. pp. 431–445 (2004)