19

Causal and Probabilistic Reasoning in P-log

MICHAEL GELFOND AND NELSON RUSHTON

1 Introduction

In this paper we give an overview of the knowledge representation (KR) language P-log [Baral, Gelfond, and Rushton 2009] whose design was greatly influenced by work of Judea Pearl. We introduce the syntax and semantics of P-log, give a number of examples of its use for knowledge representation, and discuss the role Pearl's ideas played in the design of the language. Most of the technical material presented in the paper is not new. There are however two novel technical contributions which could be of interest. First we expand P-log semantics to allow domains with infinite Herbrand bases. This allows us to represent infinite sequences of random variables and (indirectly) continuous random variables. Second we generalize the logical base of P-log which improves the degree of elaboration tolerance of the language.

The goal of the P-log designers was to create a KR-language allowing natural and elaboration tolerant representation of commonsense knowledge involving logic and probabilities. The logical framework of P-log is Answer Set Prolog (ASP) — a language for knowledge representation and reasoning based on the answer set semantics (*aka* stable model semantics) of logic programs [Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991]. ASP has roots in declarative programing, the syntax and semantics of standard Prolog, disjunctive databases, and non-monotonic logic. The semantics of ASP captures the notion of possible beliefs of a reasoner who adheres to the *rationality principle* which says that "One shall not believe anything one is not forced to believe". The entailment relation of ASP is non-monotonic[1], which facilitates a high degree of elaboration tolerance in ASP theories. ASP allows natural representation of defaults and their exceptions, causal relations (including effects of actions), agents' intentions and obligations, and other constructs of natural language. ASP has a number of efficient reasoning systems, a well developed mathematical theory, and a well tested methodology of representing and using knowledge for computational tasks (see, for instance, [Baral 2003]). This, together with the fact that some of the designers of P-log came from the ASP community made the choice of a logical foundation for P-log comparatively easy.

The choice of a probabilistic framework was more problematic and that is where Judea's ideas played a major role. Our first problem was to choose from among various conceptualizations of probability: classical, frequentist, subjective, etc. Understanding the intuitive

[1] Roughly speaking, a language L is *monotonic* if whenever Π_1 and Π_2 are collections of statements of L with $\Pi_1 \subset \Pi_2$, and W is a model of Π_2, then W is a model of Π_1. A language which is not monotonic is said to be *nonmonotonic*.

readings of basic language constructs is crucial for a software/knowledge engineer — probably more so than for a mathematician who may be primarily interested in their mathematical properties. Judea Pearl in [Pearl 1988] introduced the authors to the subjective view of probability — i.e. understanding of probabilities as degrees of belief of a rational agent — and to the use of subjective probability in AI. This matched well with the ASP-based logic side of the language. The ASP part of a P-log program can be used for describing possible beliefs, while the probabilistic part would allow knowledge engineers to quantify the degrees of these beliefs.

After deciding on an intuitive reading of probabilities, the next question was *which sorts of probabilistic statements to allow*. Fortunately, the question of concise and transparent representation of probability distributions was already addressed by Judea in [Pearl 1988], where he showed how Bayesian nets can be successfully used for this purpose. The concept was extended in [Pearl 2000] where Pearl introduced the notion of Causal Bayesian Nets (CBN's). Pearl's definition of CBN's is pioneering in three respects. First, he gives a framework where nondeterministic causal relations are the primitive relations among random variables. Second, he shows how relationships of correlation and (classical) independence *emerge* from these causal relationships in a natural way; and third he shows how this emergence is faithful to our intuitions about the difference between causality and (mere) correlation.

As we mentioned above, one of the primary desired features in the design of P-log was elaboration tolerance — defined as the ability of a representation to incorporate new knowledge with minimal revision [McCarthy 1999]. P-log inherited from ASP the ability to naturally incorporate many forms of new logical knowledge. An extension of ASP, called CR-Prolog, further improved this ability [Balduccini and Gelfond 2003]. The term "elaboration tolerance" is less well known in the field of probabilistic reasoning, but one of the primary strengths of Bayes nets as a representation is the ability to systematically and smoothly incorporate new knowledge through conditioning, using Bayes Theorem as well as algorithms given by Pearl [Pearl 1988] and others. Causal Bayesian Nets carry this a step further, by allowing us to formalize interventions in addition to (and as distinct from) observations, and smoothly incorporate either kind of new knowledge in the form of updates. Thus from the standpoint of elaboration tolerance, CBN's were a natural choice as a probabilistic foundation for P-log.

Another reason for choosing CBN's is that we simply believe Pearl's distinction between observations and interventions to be central to commonsense probabilistic reasoning. It gives a precise mathematical basis for distinguishing between the following questions: (1) what can I expect to happen given that I observe $X = x$, and (2) what can I expect to happen if I *intervene in the normal operation of a probabilistic system* by fixing value of variable X to x? These questions could in theory be answered using classical methods, but only by creating a separate probabilistic model for each question. In a CBN these two questions may be treated as conditional probabilities (one conditioned on an observation and the other on an action) of a single probabilistic model.

P-log carries things another step. There are many actions one could take to manipulate a

system besides fixing the values of (otherwise random) variables — and the effects of such actions are well studied under headings associated with ASP. Moreover, besides actions, there are many sorts of information one might gain besides those which simply eliminate possible worlds: one may gain knowledge which introduces new possible worlds, alters the probabilities of possible worlds, introduces new logical rules, etc. ASP has been shown to be a good candidate for handling such updates in non-probabilistic settings, and our hypothesis was that it would serve as well when combined with a probabilistic representation. Thus some of the key advantages of Bayesian nets, which are amplified by CBN's, show plausible promise of being even further amplified by their combination with ASP. This is the methodology of P-log: to combine a well studied method for elaboration tolerant probabilistic representations (CBN's) with a well studied method for elaboration tolerant logical representations (ASP).

Finally let us say a few words about the current status of the language. It is comparatively new. The first publication on the subject appeared in [Baral, Gelfond, and Rushton 2004], and the full journal paper describing the language appeared only recently in [Baral, Gelfond, and Rushton 2009]. The use of P-log for knowledge representation was also explored in [Baral and Hunsaker 2007] and [Gelfond, Rushton, and Zhu 2006]. A prototype reasoning system based on ASP computation allowed the use of the language for a number of applications (see, for instance, [Baral, Gelfond, and Rushton 2009; Pereira and Ramli 2009]). We are currently working on the development and implementation of a more efficient system, and on expanding it to allow rules of CR-Prolog. Finding ways for effectively combining ASP-based computational methods of P-log with recent advanced algorithms for Bayesian nets is probably one of the most interesting open questions in this area.

The paper is organized as follows. Section 2 contains short introduction to ASP and CR-Prolog. Section 3 describes the syntax and informal semantics of P-log, illustrating both through a nontrivial example. Section 4 gives another example, similar in nature to Simpson's Paradox. Section 5 states a new theorem which extends the semantics of P-log from that given in [Baral, Gelfond, and Rushton 2009] to cover programs with infinitely many random variables. The basic idea of Section 5 is accessible to a general audience, but its technical details require an understanding of the material presented in [Baral, Gelfond, and Rushton 2009].

2 Preliminaries

This section contains a description of syntax and semantics of both ASP and CR-Prolog. In what follows we use a standard notion of a sorted signature from classical logic. Terms and atoms are defined as usual. An atom $p(\bar{t})$ and its negation $\neg p(\bar{t})$ are referred to as *literals*. Literals of the form $p(\bar{t})$ and $\neg p(\bar{t})$ are called *contrary*. ASP and CR-Prolog also contain connectives *not* and *or* which are called *default negation* and *epistemic disjunction* respectively. Literals possibly preceded by default negation are called *extended literals*.

An ASP program is a pair consisting of a signature σ and a collection of rules of the form

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

where l's are literals. The right-hand side of of the rule is often referred to as the rule's *body*, the left-hand side as the rule's head.

The answer set semantics of a logic program Π assigns to Π a collection of *answer sets* – partial interpretations[2] corresponding to possible sets of beliefs which can be built by a rational reasoner on the basis of rules of Π. In the construction of such a set S, the reasoner is assumed to be guided by the following informal principles:

- S must satisfy the rules of Π;
- the reasoner should adhere to the *rationality principle*, which says that *one shall not believe anything one is not forced to believe*.

To understand the former let us consider a partial interpretation S viewed as a possible set of beliefs of our reasoner. A ground atom p is satisfied by S if $p \in S$, i.e., the reasoner believes p to be true. According to the semantics of our connectives $\neg p$ means that p is false. Consequently, $\neg p$ is satisfied by S iff $\neg p \in S$, i.e., the reasoner believes p to be false. Unlike $\neg p$, $not \ p$ has an epistemic character and is read as *there is no reason to believe that p is true*. Accordingly, S satisfies $not \ l$ if $l \notin S$. (Note that it is possible for the reasoner to believe neither p nor $\neg p$). An epistemic disjunction $l_1 \ or \ l_2$ is satisfied by S if $l_1 \in S$ or $l_2 \in S$, i.e., the reasoner believes at least one of the disjuncts to be true. Finally, S satisfies the body (resp., head) of rule (1) if S satisfies all of the extended literals occurring in its body (resp., head); and S satisfies rule (1) if S satisfies its head or does not satisfy its body.

What is left is to capture the intuition behind the rationality principle. This will be done in two steps.

DEFINITION 1 (Answer Sets, Part I). Let program Π consist of rules of the form:

$$l_0 \ or \ \ldots \ or \ l_i \leftarrow l_{i+1}, \ldots, l_m.$$

An answer set of Π is a consistent set S of ground literals such that:

- S satisfies the rules of Π.
- S is minimal; i.e., no proper subset of S satisfies the rules of Π.

The rationality principle here is captured by the minimality condition. For example, it is easy to see that $\{\ \}$ is the only answer set of program consisting of the single rule $p \leftarrow p$, and hence the reasoner associated with it knows nothing about the truth or falsity of p. The program consisting of rules

[2] By partial interpretation we mean a consistent set of ground literals of $\sigma(\Pi)$.

$p(a)$.
$q(a)$ *or* $q(b) \leftarrow p(a)$.

has two answer sets: $\{p(a), q(a)\}$ and $\{p(a), q(b)\}$. Note that no rule requires the reasoner to believe in both $q(a)$ and $q(b)$. Hence he believes that the two formulas $p(a)$ and $(q(a)$ *or* $q(b))$ are true, and that $\neg p(a)$ is false. He remains undecided, however, about, say, the two formulas $p(b)$ and $(\neg q(a)$ *or* $\neg q(b))$. Now let us consider an arbitrary program:

DEFINITION 2 (Answer Sets, Part II). Let Π be an arbitrary collection of rules (1) and S a set of literals. By Π^S we denote the program obtained from Π by

1. removing all rules containing $not\ l$ such that $l \in S$;
2. removing all other premises containing not .

S is an answer set of Π iff S is an answer set of Π^S.

To illustrate the definition let us consider a program

$p(a)$.
$p(b)$.
$\neg p(X) \leftarrow not\ p(X)$.

where p is a unary predicate whose domain is the set $\{a, b, c\}$. The last rule, which says that if X is not believed to satisfy p then $p(X)$ is false, is the ASP formalization of a Closed World Assumption for a relation p [Reiter 1978]. It is easy to see that $\{p(a), p(b), \neg p(c)\}$ is the only answer set of this program. If we later learn that c satisfies p, this information can be simply added to the program as $p(c)$. The default for c will be defeated and the only answer set of the new program will be $\{p(a), p(b), p(c)\}$.

The next example illustrates the ASP formalization of a more general default. Consider a statement: "Normally, computer science courses are taught only by computer science professors. The logic course is an exception to this rule. It may be taught by faculty from the math department." This is a typical *default* with a *weak exception*[3] which can be represented in ASP by the rules:

$$\begin{array}{lcl} \neg may\_teach(P, C) & \leftarrow & \neg member(P, cs), \\ & & course(C, cs), \\ & & not\ ab(d_1(P, C)), \\ & & not\ may\_teach(P, C). \\ ab(d_1(P, logic)) & \leftarrow & not\ \neg member(P, math). \end{array}$$

Here $d_1(P, C)$ is the name of the default rule and $ab(d_1(P, C))$ says that default $d_1(P, C)$ is not applicable to the pair $\langle P, C\rangle$. The second rule above stops the application of the default in cases where the class is $logic$ and P *may be* a math professor. Used in conjunction with rules:

[3] An exception to a default is called *weak* if it stops application of the default without defeating its conclusion. Otherwise it is called *strong*.

$member(john, cs)$.
$member(mary, math)$.
$member(bob, ee)$.
$\neg member(P, D) \leftarrow not\ member(P, D)$.
$course(logic, cs)$.
$course(data\_structures, cs)$.

the program will entail that Mary does not teach data structures while she may teach logic; Bob teaches neither logic nor data structures, and John may teach both classes.

The previous examples illustrate the representation of defaults and their strong and weak exceptions. There is another type of possible exception to defaults, sometimes referred to as an **indirect exception**. Intuitively, these are rare exceptions that come into play only as a last resort, to restore the consistency of the agent's world view when all else fails. The representation of indirect exceptions seems to be beyond the power of ASP. This observation led to the development of a simple but powerful extension of ASP called **CR-Prolog** (or ASP with consistency-restoring rules). To illustrate the problem let us consider the following example.

Consider an ASP representation of the default "elements of class c normally have property p":

$$\begin{array}{rcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \end{array}$$

together with the rule

$$q(X) \quad \leftarrow \quad p(X).$$

and the facts $c(a)$ and $\neg q(a)$. Let us denote this program by E, where E stands for "exception".

It is not difficult to check that E is inconsistent. No rules allow the reasoner to prove that the default is not applicable to a (i.e. to prove $ab(d(a))$) or that a does not have property p. Hence the default must conclude $p(a)$. The second rule implies $q(a)$ which contradicts one of the facts. However, there seems to exists a commonsense argument which may allow a reasoner to avoid inconsistency, and to conclude that a is an indirect exception to the default. The argument is based on the **Contingency Axiom** for default $d(X)$ which says that *any element of class c can be an exception to the default $d(X)$ above, but such a possibility is very rare, and, whenever possible, should be ignored.* One may informally argue that since the application of the default to a leads to a contradiction, the possibility of a being an exception to $d(a)$ cannot be ignored and hence a must satisfy this rare property.

In what follows we give a brief description of CR-Prolog — an extension of ASP capable of encoding and reasoning about such rare events.

A program of CR-Prolog is a four-tuple consisting of

1. A (possibly sorted) signature.

2. A collection of regular rules of ASP.

3. A collection of rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, not\ l_{k+1}, \ldots, not\ l_n \tag{2}$$

   where l's are literals. Rules of this type are called *consistency restoring* rules (CR-rules).

4. A partial order, $\leq$, defined on sets of CR-rules. This partial order is often referred to as a **preference relation**.

Intuitively, rule (2) says that if the reasoner associated with the program believes the body of the rule, then he "may possibly" believe its head. However, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program. The partial order over sets of CR-rules will be used to select preferred possible resolutions of the conflict. Currently the inference engine of CR-Prolog [Balduccini 2007] supports two such relations, denoted $\leq_1$ and $\leq_2$. One is based on the set-theoretic inclusion ($R_1 \leq_1 R_2$ holds iff $R_1 \subseteq R_2$). The other is defined by the cardinality of the corresponding sets ($R_1 \leq_2 R_2$ holds iff $|R_1| \leq |R_2|$). To give the precise semantics we will need some terminology and notation.

The set of regular rules of a CR-Prolog program Π will be denoted by Π^r, and the set of CR-rules of Π will be denoted by Π^{cr}. By $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$. If R is a set of CR-rules then $\alpha(R) = \{\alpha(r) \ : \ r \in R\}$. As in the case of ASP, the semantics of CR-Prolog will be given for ground programs. A rule with variables will be viewed as a shorthand for a set of ground rules.

DEFINITION 3. (Abductive Support)
A minimal (with respect to the preference relation of the program) collection R of CR-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an **abductive support** of Π.

DEFINITION 4. (Answer Sets of CR-Prolog)
A set A is called an *answer set* of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

Now let us show how CR-Prolog can be used to represent defaults and their indirect exceptions. The CR-Prolog representation of the default $d(X)$, which we attempted to represent in ASP program E, may look as follows

$$\begin{array}{rcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \\ \neg p(X) & \stackrel{+}{\leftarrow} & c(X). \end{array}$$

The first rule is the standard ASP representation of the default, while the second rule expresses the Contingency Axiom for the default $d(X)$[4]. Consider now a program obtained by combining these two rules with an atom $c(a)$.

Assuming that a is the only constant in the signature of this program, the program's unique answer set will be $\{c(a), p(a)\}$. Of course this is also the answer set of the regular part of our program. (Since the regular part is consistent, the Contingency Axiom is ignored.) Let us now expand this program by the rules

$q(X) \leftarrow p(X).$
$\neg q(a).$

The regular part of the new program is inconsistent. To save the day we need to use the Contingency Axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set $\{\neg q(a), c(a), \neg p(a))\}$. The new information does not produce inconsistency, as it did in ASP program E. Instead the program withdraws its previous conclusion and recognizes a as a (strong) exception to default $d(a)$.

3 The Language

A P-log program consists of its *declarations*, *logical rules*, *random selection rules*, *probability atoms*, *observations*, and *actions*. We will begin this section with a brief description of the syntax and informal readings of these components of the programs, and then proceed to an illustrative example.

The declarations of a P-log program give the types of objects and functions in the program. Logical rules are "ordinary" rules of the underlying logical language written using light syntactic sugar. For purposes of this paper, the underlying logical language is CR-Prolog.

P-log uses *random selection rules* to declare random attributes (essentially random variables) of the form $a(\bar{t})$, where a is the name of the attribute and $\bar{t}$ is a vector of zero or more parameters. In this paper we consider random selection rules of the form

$$[\ r\ ]\ random(a(\bar{t})) \leftarrow B. \tag{3}$$

where r is a term used to name the random causal process associated with the rule and B is a conjunction of zero or more extended literals. The name $[\ r\ ]$ is optional and can be omitted if the program contains exactly one random selection rule for $a(\bar{t})$. Statement (3) says that *if B were to hold, the value of $a(\bar{t})$ would be selected at random from its range by process r, unless this value is fixed by a deliberate action*. More general forms of random selection rules, where the values may be selected from a range which depends on context, are discussed in [Baral, Gelfond, and Rushton 2009].

[4] In this form of Contingency Axiom, we treat X as a strong exception to the default. Sometimes it may be useful to also allow weak indirect exceptions; this can be achieved by adding the rule $ab(d(X)) \stackrel{+}{\leftarrow} c(X)$.

Knowledge of the numeric probabilities of possible values of random attributes is expressed through *causal probability atoms*, or pr-atoms. A pr-atom takes the form

$$pr_r(a(\bar{t}) = y|_c\ B) = v$$

where $a(\bar{t})$ is a random attribute, B a conjunction of literals, r is a causal process, $v \in [0, 1]$, and y is a possible value of $a(\bar{t})$. The statement says that *if the value of $a(\bar{t})$ is fixed by process r, and B holds, then the probability that r causes $a(\bar{t}) = y$ is v.* If r is uniquely determined by the program then it can be omitted. The "causal stroke" '$|_c$' and the "rule body" B may also be omitted in case B is empty.

Observations and actions of a P-log program are written, respectively, as

$$obs(l). \qquad do(a(\bar{t}) = y)).$$

where l is a literal, $a(\bar{t})$ a random attribute, and y a possible value of $a(\bar{t})$. $obs(l)$ is read *l is observed to be true*. The action $do(a(\bar{t}) = y)$ is read *the value of $a(\bar{t})$, instead of being random, is set to y by a deliberate action*.

This completes a general introductory description of P-log. Next we give an example to illustrate this description. The example shows how certain forms of knowledge may be represented, including deterministic causal knowledge, probabilistic causal knowledge, and strict and defeasible logical rules (a rule is *defeasible* if it states an overridable presumption; otherwise it is *strict*). We will use this example to illustrate the syntax of P-log, and, afterward, to provide an indication of the formal semantics. Complete syntax and semantics are given in [Baral, Gelfond, and Rushton 2009], and the reader is invited to refer there for more details.

EXAMPLE 5. [Circuit]
A circuit has a motor, a breaker, and a switch. The switch may be open or closed. The breaker may be tripped or not; and the motor may be turning or not. The operator may toggle the switch or reset the breaker. If the switch is closed and the system is functioning normally, the motor turns. The motor never turns when the switch is open, the breaker is tripped, or the motor is burned out. The system may break and if so the break could consist of a tripped breaker, a burned out motor, or both, with respective probabilities .9, .09, and .01. Breaking, however, is rare, and should be considered only in the absence of other explanations.

Let us show how to represent this knowledge in P-log. First we give declarations of sorts and functions relevant to the domain. As typical for representation of dynamic domains we will have sorts for actions, fluents (properties of the domain which can be changed by actions), and time steps. Fluents will be partitioned into inertial fluents and defined fluents. The former are subject to the law of inertia [Hayes and McCarthy 1969] (which says that things stay the same by default), while the latter are specified by explicit definitions in terms of already defined fluents. We will also have a sort for possible types of breaks which may occur in the system. In addition to declared sorts P-log contains a number of predefined sorts, e.g. a sort *boolean*. Here are the sorts of the domain for the circuit example:

$action = \{toggle, reset, break\}.$

$inertial\_fluent = \{closed, tripped, burned\}.$

$defined\_fluent = \{turning, faulty\}.$

$fluent = inertial\_fluent \cup defined\_fluent.$

$step = \{0, 1\}.$

$breaks = \{trip, burn, both\}.$

In addition to sorts we need to declare functions (referred in P-log as *attributes*) relevant to our domain.

$holds\ :\ fluent \times step \rightarrow boolean.$

$occurs\ :\ action \times step \rightarrow boolean.$

Here $holds(f, T)$ says that fluent f is true at time step T and $occurs(a, T)$ indicates that action a was executed at T.

The last function we need to declare is a random attribute $type\_of\_break(T)$ which denotes the type of an occurrence of action $break$ at step T.

$type\_of\_break\ :\ step \rightarrow breaks.$

The first two logical rules of the program define the direct effects of action $toggle$.

$$\begin{array}{lcl} holds(closed, T+1) & \leftarrow & occurs(toggle, T), \\ & & \neg holds(closed, T). \\ \neg holds(closed, T+1) & \leftarrow & occurs(toggle, T), \\ & & holds(closed, T). \end{array}$$

They simply say that toggling opens and closes the switch. The next rule says that resetting the breaker untrips it.

$$\neg holds(tripped, T+1) \quad \leftarrow \quad occurs(reset, T).$$

The effects of action $break$ are described by the rules

$$\begin{array}{lcl} holds(tripped, T+1) & \leftarrow & occurs(break, T), \\ & & type\_of\_break(T) = trip. \\ holds(burned, T+1) & \leftarrow & occurs(break, T), \\ & & type\_of\_break(T) = burn. \\ holds(tripped, T+1) & \leftarrow & occurs(break, T), \\ & & type\_of\_break(T) = both. \\ holds(burned, T+1) & \leftarrow & occurs(break, T), \\ & & type\_of\_break(T) = both. \end{array}$$

The next two rules express the inertia axiom which says that *by default, things stay as they are.* They use default negation *not* — the main nonmonotonic connective of ASP —, and can be viewed as typical representations of defaults in ASP and its extensions.

$$\begin{array}{lcl} holds(F, T+1) & \leftarrow & inertial\_fluent(F), \\ & & holds(F, T), \\ & & not\ \neg holds(F, T+1). \\ \neg holds(F, T+1) & \leftarrow & inertial\_fluent(F), \\ & & \neg holds(F, T), \\ & & not\ holds(F, T+1). \end{array}$$

Next we explicitly define fluents *faulty* and *turning*.

$$\begin{array}{lcl} holds(faulty, T) & \leftarrow & holds(tripped, T). \\ holds(faulty, T) & \leftarrow & holds(burned, T). \\ \neg holds(faulty, T) & \leftarrow & not\ holds(faulty, T). \end{array}$$

The rules above say that the system is functioning abnormally if and only if the breaker is tripped or the motor is burned out. Similarly the next definition says that the motor turns if and only if the switch is closed and the system is functioning normally.

$$\begin{array}{lcl} holds(turning, T) & \leftarrow & holds(closed, T), \\ & & \neg holds(faulty, T). \\ \neg holds(turning, T) & \leftarrow & not\ holds(turning, T). \end{array}$$

The above rules are sufficient to define causal effects of actions. For instance if we assume that at Step 0 the motor is turning and the breaker is tripped, i.e. action $break$ of the type $trip$ occurred at 0, then in the resulting state we will have $holds(tripped, 1)$ as the direct effect of this action; while $\neg holds(turning, 1)$ will be its indirect effect[5].

We will next have a default saying that for each action A and time step T, in the absence of a reason to believe otherwise we assume A does not occur at T.

$\neg occurs(A, T) \leftarrow action(A), not\ occurs(A, T).$

We next state a CR-rule representing possible exceptions to this default. The rule says that a break to the system may be considered if necessary (that is, necessary in order to reach a consistent set of beliefs).

$occurs(break, 0) \stackrel{+}{\leftarrow}$.

The next collection of facts describes the initial situation of our story.

[5]It is worth noticing that, though short, our formalization of the circuit is non-trivial. It is obtained using the general methodology of representing dynamic systems modeled by transition diagrams whose nodes correspond to physically possible states of the system and whose arcs are labeled by actions. A transition $\langle \sigma_0, a, \sigma_1 \rangle$ indicates that state σ_1 may be a result of execution of a in σ_0. The problem of finding concise and mathematically accurate description of such diagrams has been a subject of research for over 30 years. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents. An additional level of complexity is added by the need to specify what is not changed by actions. As noticed by John McCarthy, the latter, known as the Frame Problem, can be reduced to finding a representation of the Inertia Axiom which requires the ability to represent defaults and to do non-monotonic reasoning. The representation of this axiom as well as that of the interrelations between fluents we used in this example is a simple special case of general theory of action and change based on logic programming under the answer set semantics.

$\neg holds(closed, 0).\ \ \neg holds(burned, 0).\ \ \neg holds(tripped, 0).\ \ occurs(toggle, 0).$

Next, we state a random selection rule which captures the non-determinism in the description of our circuit.

$random(type\_of\_break(T)) \leftarrow occurs(break, T).$

The rule says that if action $break$ occurs at step T then the type of break will be selected at random from the range of possible types of breaks, unless this type is fixed by a deliberate action. Intuitively, $break$ can be viewed as a non-deterministic action, with non-determinism coming from the lack of knowledge about the precise type of $break$.

Let π_0 be the circuit program given so far. Next we will give a sketch of the formal semantics of P-log, using π_0 as an illustrative example.

The *logical part* of a P-log program Π consists of its declarations, logical rules, random selection rules, observations, and actions; while its *probabilistic part* consists of its pr-atoms (though the above program does not have any). The semantics of P-log describes a translation of the logical part of Π into an "ordinary" CR-Prolog program $\tau(\Pi)$. The semantics of Π is then given by

1. a collection of answer sets of $\tau(\Pi)$ viewed as the set of possible worlds of a rational agent associated with Π, along with
2. a probability measure over these possible worlds, determined by the collection of the probability atoms of Π.

To obtain $\tau(\pi_0)$ we represent sorts as collections of facts. For instance, sort $step$ would be represented in CR-Prolog as

$step(0).\ \ step(1).$

For a non-boolean function $type\_of\_break$ the occurrences of atoms of the form $type\_of\_break(T) = trip$ in π_0 are replaced by $type\_of\_break(T, trip)$. Similarly for $burn$ and $both$. The translation also contains the axiom

$$\begin{array}{rcl} \neg type\_of\_break(T, V_1) & \leftarrow & breaks(V_1), breaks(V_2), V_1 \neq V_2, \\ & & type\_of\_break(T, V_2). \end{array}$$

to guarantee that $type\_of\_break$ is a function. In general, the same transformation is performed for all non-boolean functions.

Logical rules of π_0 are simply inserted into $\tau(\pi_0)$. Finally, the random selection rule is transformed into

$$\begin{array}{l} type\_of\_break(T, trip)\ or\ type\_of\_break(T, burn)\ or\ type\_of\_break(T, both) \leftarrow \\ \qquad\qquad occurs(break, T), \\ \qquad\qquad not\ intervene(type\_of\_break(T)). \end{array}$$

It is worth pointing out here that while CBN's represent the notion of intervention in terms of transformations on graphs, P-log axiomatizes the semantics of intervention by including

$not\ intervene(\dots)$ in the body of the translation of each random selection rule. This amounts to a *default presumption* of randomness, overridable by intervention. We will see next how actions using do can defeat this presumption.

Observations and actions are translated as follows. For each literal l in π_0, $\tau(\pi_0)$ contains the rule

$\leftarrow obs(l), not\ l.$

For each atom $a(\bar{t}) = y$, $\tau(\pi)$ contains the rules

$a(\bar{t}, y) \leftarrow do(a(\bar{t}, y)).$

and

$intervene(a(\bar{t})) \leftarrow do(a(\bar{t}, Y)).$

The first rule eliminates possible worlds of the program failing to satisfy l. The second rule makes sure that interventions affect their intervened-upon variables in the expected way. The third rule defines the relation $intervene$ which, for each action, cancels the randomness of the corresponding attribute.

It is not difficult to check that under the semantics of CR-Prolog, $\tau(\pi_0)$ has a unique possible world W containing $holds(closed, 1)$ and $holds(turning, 1)$, the direct and indirect effects, respectively, of the action $close$. Note that the collection of regular ASP rules of $\tau(\pi_0)$ is consistent, i.e., has an answer set. This means that CR-rule $occurs(break, 0) \stackrel{+}{\leftarrow}$ is not activated, break does not occur, and the program contains no randomness.

Now we will discuss how probabilities are computed in P-log. Let Π be a P-log program containing the random selection rule $[r]\ random(a(\bar{t})) \leftarrow B_1$ and the pr-atom $pr_r(a(\bar{t}) = y \mid_c B_2) = v$. Then if W is a possible world of Π satisfying B_1 and B_2, the *assigned probability* of $a(\bar{t}) = y$ in W is defined [6] to be v. In case W satisfies B_1 and $a(\bar{t}) = y$, but there is no pr-atom $pr_r(a(\bar{t} = y \mid_c B_2) = v$ of Π such that W satisfies B_2, then the *default probability* of $a(\bar{t}) = y$ in W is computed using the "indifference principle", which says that two possible values of a random selection are equally likely if we have no reason to prefer one to the other (see [Baral, Gelfond, and Rushton 2009] for details). The *probability* of each random atom $a(\bar{t}) = y$ occurring in each possible world W of program Π, written $P_\Pi(W, a(\bar{t}) = y)$, is now defined to be the assigned probability or the default probability, as appropriate.

Let W be a possible world of Π. The *unnormalized probability*, $\hat{\mu}_\Pi(W)$, of a possible world W *induced by* Π is

$$\hat{\mu}_\Pi(W) =_{def} \prod_{a(\bar{t},y) \in\ W} P_\Pi(W, a(\bar{t}) = y)$$

where the product is taken only over atoms for which $P(W, a(\bar{t}) = y)$ is defined.

[6] For the sake of well definiteness, we consider only programs in which at most one v satisfies this definition.

Suppose Π is a P-log program having at least one possible world with nonzero unnormalized probability, and let Ω be the set of possible worlds of Π. The *measure*, $\mu_\Pi(W)$, of a possible world W *induced by* Π is the unnormalized probability of W divided by the sum of the unnormalized probabilities of all possible worlds of Π, i.e.,

$$\mu_\Pi(W) =_{def} \frac{\hat{\mu}_\Pi(W)}{\sum_{W_i \in \Omega} \hat{\mu}_\Pi(W_i)}$$

When the program Π is clear from context we may simply write $\hat{\mu}$ and μ instead of $\hat{\mu}_\Pi$ and μ_Π respectively.

This completes the discussion of how probabilities of possible worlds are defined in P-log. Now let us return to the circuit example. Let program π_1 be the union of π_0 with the single observation

$obs(\neg holds(turning, 1))$

The observation contradicts our previous conclusion $holds(turning, 1)$ reached by using the effect axiom for $toggle$, the definitions of $faulty$ and $turning$, and the inertia axiom for $tripped$ and $burned$. The program $\tau(\pi_1)$ will resolve this contradiction by using the CR-rule $occurs(break, 0) \stackrel{+}{\leftarrow}$ to conclude that the action $break$ occurred at Step 0. Now $type\_of\_break$ randomly takes one of its possible values. Accordingly, $\tau(\pi_1)$ has three answer sets: W_1, W_2, and W_3. All of them contain $occurs(break, 0)$, $holds(faulty, 1)$, $\neg holds(turning, 1)$. One, say W_1 will contain

$type\_of\_break(1, trip)$, $holds(tripped, 1)$, $\neg holds(burned, 1)$

W_2 and W_3 will respectively contain

$type\_of\_break(1, burn)$, $\neg holds(tripped, 1)$, $holds(burned, 1)$

and

$type\_of\_break(1, both)$, $holds(tripped, 1)$, $holds(burned, 1)$

In accordance with our general definition, π_1 will have three possible worlds, W_1, W_2, and W_3. The probabilities of each of these three possible worlds can be computed as $1/3$, using the indifference principle.

Now let us add some quantitative probabilities to our program. If π_2 is the union of π_1 with the following three pr-atoms

$pr(type\_of\_break(T) = trip \mid_c break(T)) = 0.9$
$pr(type\_of\_break(T) = burned \mid_c break(T)) = 0.09$
$pr(type\_of\_break(T) = both \mid_c break(T)) = 0.01$

then program π_2 has the same possible worlds as Π_1. Not surprisingly, $P_{\pi_2}(W_1) = 0.9$. Similarly $P_{\pi_2}(W_2) = 0.09$ and $P_{\pi_2}(W_3) = 0.01$. This demonstrates how a P-log program may be written in stages, with quantitative probabilities added as they are needed or become available.

Typically we are interested not just in the probabilities of individual possible worlds, but in the probabilities of certain interesting sets of possible worlds described, e.g., those described by formulae. For current purposes a rather simple definition suffices. Viz., recalling that possible worlds are sets of literals, for an arbitrary set C of literals we define

$$P_\pi(C) =_{def} P_\pi(\{W : C \subseteq W\}).$$

For example, $P_{\pi_1}(holds(turning, 1)) = 0$, $P_{\pi_1}(holds(tripped, 1)) = 1/3$, and $P_{\pi_2}(holds(tripped, 1)) = 0.91$.

Our example is in some respects rather simple. For instance, every possible world of our program contains at most one atom of the form $a(\bar{t}) = y$ where $a(\bar{t})$ is a random attribute. We hope, however, that this example gives a reader some insight in the syntax and semantics of P-log. It is worth noticing that the example shows the ability of P-log to mix logical and probabilistic reasoning, including reasoning about causal effects of actions and explanations of observations. In addition it demonstrates the non-monotonic character of P-log, i.e. its ability to react to new knowledge by changing probabilistic models of the domain and creating new possible worlds.

The ability to introduce new possible worlds as a result of conditioning is of interest from two standpoints. First, it reflects the common sense semantics of utterances such as "the motor might be burned out." Such a sentence does not eliminate existing possible beliefs, and so there is no classical (i.e., monotonic) semantics in which the statement would be informative. If it is informative, as common sense suggests, then its content seems to introduce new possibilities into the listener's thought process.

Second, nonmonotonicity can improve performance. Possible worlds tend to proliferate exponentially with the size of a program, quickly making computations intractable. The ability to consider only those random selections which may explain our abnormal observations may make computations tractable for larger programs. Even though our current solver is in its early stages of development, it is based on well researched answer set solvers which efficiently eliminate impossible worlds from consideration based on logical reasoning. Thus even our early prototype has shown promising performance on problems where logic may be used to exclude possible worlds from consideration in the computation of probabilities [Gelfond, Rushton, and Zhu 2006].

4 Spider Example

In this section, we consider a variant of Simpson's paradox, to illustrate the formalization of interventions in P-log. The story we would like to formalize is as follows:

In Stan's home town there are two kinds of poisonous spider, the creeper and the spinner. Bites from the two are equally common in Stan's area — though spinner bites are more common on a worldwide basis. An experimental anti-venom has been developed to treat bites from either kind of spider, but its effectiveness is questionable.

One morning Stan wakes to find he has a bite on his ankle, and drives to the emergency room. A doctor examines the bite, and concludes it is a bite from either a creeper or a

spinner. In deciding whether to administer the anti-venom, the doctor examines the data he has on bites from the two kinds of spiders: out of 416 people bitten by the creeper worldwide, 312 received the anti-venom and 104 did not. Among those who received the anti-venom, 187 survived; while 73 survived who did not receive anti-venom. The spinner is more deadly and tends to inhabit areas where the treatment is less available. Of 924 people bitten by the spinner, 168 received the anti-venom, 34 of whom survived. Of the 756 spinner bite victims who did not receive the experimental treatment, only 227 survived.

For a random individual bitten by a creeper or spinner, let s, a, and c denote the events of *survival, administering anti-venom,* and *creeper bite.* Based on the fact that the two sorts of bites are equally common in Stan's region, the doctor assigns a 0.5 probability to either kind of bite. He also computes a probability of survival, with and without treatment, from each kind of bite, based on the sampling distribution of the available data. He similarly computes the probabilities that victims of each kind of bite received the anti-venom. We may now imagine the doctor uses Bayes' Theorem to compute $P(s \mid a) = 0.522$ and $P(s \mid \neg a) = 0.394$.

Thus we see that if we choose a historical victim, in such a way that he has a 50/50 chance of either kind of bite, those who received anti-venom would have a substantially higher chance of survival. Stan is in the situation of having a 50/50 chance of either sort of bite; however, he is *not* a historical victim. Since we are intervening in the decision of whether he receives anti-venom, the computation above is not germane (as readers of [Pearl 2000] already know) — though we can easily imagine the doctor making such a mistake. A correct solution is as follows. Formalizing the relevant parts of the story in a P-log program Π gives

$survive, antivenom \ : \ boolean.$
$spider \ : \ \{creeper, spinner\}.$

$random(spider).$
$random(survive).$
$random(antivenom).$

$pr(spider = creeper) = 0.5.$

$pr(survive \mid_c spider = creeper, antivenom) = 0.6.$
$pr(survive \mid_c spider = creeper, \neg antivenom) = 0.7.$
$pr(survive \mid_c spider = spinner, antivenom) = 0.2.$
$pr(survive \mid_c spider = spinner, \neg antivenom) = 0.3.$

and so, according to our semantics,

$P_{\Pi \cup \{do(antivenom\}}(survive) = 0.4$
$P_{\Pi \cup \{do(\neg antivenom\}}(survive) = 0.5$

Thus, the correct decision, assuming we want to intervene to maximize Stan's chance of survival, is to not administer antivenom.

In order to reach this conclusion by classical probability, we would need to consider separate probability measures P_1 and P_2, on the sets of patients who received or did not receive antivenom, respectively. If this is done correctly, we obtain $P_1(s) = 0.4$ and $P_2(s) = 0.5$, as in the P-log program.

Thus we can get a correct classical solution using separate probability measures. Note however, that we could also get an *incorrect* classical solution using separate measures, since there exist probability measures $\hat{P}_1$ and $\hat{P}_2$ on the sets of historical bite victims which capture classical conditional probabilities given a and $\neg a$ respectively. We may define

$\hat{P}_1(E) =_{def} \frac{P(E \cap a)}{0.3582}$

$\hat{P}_2(E) =_{def} \frac{P(E \cap \neg a)}{0.6418}$

It is well known that each of these is a probability measure. They are seldom seen only because classical conditional probability gives us simple notations for them *in terms of a single measure capturing common background knowledge*. This allows us to refer to probabilities conditioned on observations without defining a new measure for each such observation. What we do not have, classically, is a similar mechanism for probabilities conditioned on intervention — which is sometimes of interest as the example shows. The ability to condition on interventions in this way has been a fundamental contribution of Pearl; and the inclusion in P-log of such conditioning-on-intervention is a direct result of the authors' reading of his book.

5 Infinite Programs

The definitions given so far for P-log apply only to programs with finite numbers of random selection rules. In this section we state a theorem which allows us to extend these semantics to programs which may contain infinitely many random selection rules. No changes are required from the syntax given in [Baral, Gelfond, and Rushton 2009], and the probability measure described here agrees with the one in [Baral, Gelfond, and Rushton 2009] whenever the former is defined.

We begin by defining the class of programs for which the new semantics are applicable. The reader is referred to [Baral, Gelfond, and Rushton 2009] for the definitions of *causally ordered*, *unitary*, and *strict probabilistic levelling*.

DEFINITION 6. [Admissible Program]
A P-log program is *admissible* if it is causally ordered and unitary, and if there exists a strict probabilistic levelling $||$ on Π such that no ground literal occurs in the heads of rules in infinitely many Π_i with respect to $||$.

The condition of admissibility, and the definitions it relies on, are all rather involved to state precisely, but the intuition is as follows. Basically, a program is unitary if the probabilities assigned to the possible outcomes of each selection rule are either all assigned

and sum to 1, or are not all assigned and their sum does not exceed 1. The program is causally ordered if its causal dependencies are acyclic and if the only nondeterminism in it is a result of random selection rules. A strict probabilistic levelling is a well ordering of the selection rules of a program which witnesses the fact that it is causally ordered. Finally, a program which meets these conditions is admissible if every ground literal in the program logically depends on only finitely many random experiments. For example, the following program is not unitary:

$random(a) : boolean.$
$pr(a) = 1/2.$
$pr(\neg a) = 2/3.$

The following program is not causally ordered:

$random(a) : boolean.$
$random(b) : boolean.$
$pr_r(a|_c\, b) = 1/3.$
$pr_r(a|_c\, \neg b) = 2/3.$
$pr_r(b|_c\, a) = 1/5.$

and neither is the following:

$p \leftarrow not\ q.$
$q \leftarrow not\ p.$

since it has two answer sets which arise from circularity of defaults, rather than random selections. The following program is both unitary and causally ordered, but not admissible, because $atLeastOneTail$ depends on infinitely many coin tosses.

$coin\_toss : positive\_integer \rightarrow \{head, tail\}.$
$atLeastOneTail : boolean.$
$random(coin\_toss(N)).$
$atLeastOneTail \leftarrow coin\_toss(N) = tail.$

We need one more definition before stating the main theorem:

DEFINITION 7. [Cylinder algebra of Π]
Let Π be a countably infinite P-log program with random attributes $a_i(t)$, $i > 0$, and let C be the collection of sets of the form $\{\omega \ : a_i(t) = y \in \omega\}$ for arbitrary t, i, and y. The sigma algebra generated by C will be called the *cylinder algebra* of program Π.

Intuitively, the cylinder algebra of a program Π is the collection of sets which can be formed by performing countably many set operations (union, intersection, and complement) upon sets whose probabilities are defined by finite subprograms. We are now ready to state the main proposition of this section.

PROPOSITION 8. *[Admissible programs]*
Let Π *be an admissible P-log program with at most countably infinitely many ground rules, and let* A *be the cylinder algebra of* Π. *Then there exists a unique probability measure* P_Π

defined on A such that whenever $[r]\ random(a(\bar{t})) \leftarrow B_1$ and $pr_r(a(\bar{t}) = y \mid B_2) = v$ occur in Π, and $P_{\Pi}(B_1 \wedge B_2) > 0$, we have $P_{\Pi}(a(\bar{t}) = y \mid B_1 \wedge B_2) = v$.

Recall that the semantic value of a P-log program Π consists of (1) a set of possible worlds of Π and (2) a probability measure on those possible worlds. The proposition now puts us in position to give semantics for programs with infinitely many random selection rules. The possible worlds of the program are the answer sets of the associated (infinite) CR-Prolog program, as determined by the usual definition — while the probability measure is P_{Π}, as defined in Proposition 8.

We next give an example which exercises the proposition, in a form of a novel paradox. Imagine a casino which offers an infinite sequence of games, of which our agent may decide to play as many or as few as he wishes. For the n^{th} game, a fair coin is tossed n times. If the agent chooses to play the n^{th} game, then the agent wins $2^{n+1} + 1$ dollars if all tosses made in the n^{th} game are heads and otherwise loses one dollar.

We can formalize this game as an infinite P-log program Π. First, we declare a countable sequence of games and an integer valued variable, representing the player's net winnings after each game.

$game : positive\_integer.$
$winnings : game \rightarrow integer.$
$play : game \rightarrow boolean.$
$coin : \{\langle M, N\rangle \mid 1 \leq M \leq N\} \rightarrow \{head, tail\}.$

Note that the declaration for $coin$ is not written in the current syntax of P-log; but to save space we use set-builder notation here as a shorthand for the more lengthy formal declaration. Similarly, the notation $\langle M, N\rangle$ is also a shorthand. From this point on we will write $coin(M, N)$ instead of $coin(\langle M, N\rangle)$.

Π also contains a declaration to say that the throws are random and the coin is known to be fair:

$random(coin(M, N)).$
$pr(coin(M, N) = head) = 1/2.$

The conditions of winning the N^{th} game are described as follows:

$lose(N) \leftarrow play(N),\ coin(N, M) = tail.$
$win(N) \leftarrow play(N),\ not\ lose(N).$

The amount the agent wins or loses on each game is given by

$winnings(0) = 0.$
$winnings(N + 1) = winnings(N) + 1 + 2^{N+1} \leftarrow win(N).$
$winnings(N + 1) = winnings(N) - 1 \leftarrow lose(N).$
$winnings(N + 1) = winnings(N) \leftarrow \neg play(N).$

Finally the program contains rules which describe the agent's strategy in choosing which games to play. Note that the agent's expected winnings in the N^{th} game are given by $(1/2^N)(1 + 2^{N+1}) - (1 - 1/2^N) = 1$, so each game has positive expectation for the player. Thus a reasonable strategy might be to play every game, represented as

$play(N)$.

This completes program Π. It can be shown to be admissible, and hence there is a unique probability measure P_Π satisfying the conclusion of Proposition 1. Thus, for example, $P_\Pi(coin(3,2) = head) = 1/2$, and $P_\Pi(win(10)) = 1/2^{10}$. Each of these probabilities can be computed from finite sub-programs. As more interesting example, let S be the set of possible worlds in which the agent wins infinitely many games. The probability of this event cannot be computed from any finite sub-program of Π. However, S is a countable intersection of countable unions of sets whose probabilities are defined by finite subprograms. In particular,

$$S = \bigcap_{N=1}^{\infty} \bigcup_{J=N}^{\infty} \{W \mid win(J) \in W\}$$

and therefore, S is in the cylinder algebra of Π and so its probability is given by the measure defined in Proposition 1.

So where is the Paradox? To see this, let us compute the probability of S. Since P_Π is a probability measure, it is monotonic in the sense that no set has greater probability than any of its subsets. P_Π must also be *countably subadditive*, meaning that the probability of a countable union of sets cannot exceed the sum of their probabilities. Thus, from the above we get for every N,

$$\begin{aligned} P_\Pi(S) &< P_\Pi(\bigcup_{J=N}^{\infty} \{W \mid win(J) \in W\} \\ &\leq \sum_{J=N}^{\infty} P_\Pi(\{W \mid win(J) \in W\}) \\ &= \sum_{J=N}^{\infty} 1/2^J \\ &= 1/2^N \end{aligned}$$

Now since right hand side can be made arbitrarily small by choosing a sufficiently large N, it follows that $P_\Pi(S) = 0$. Consequently, with probability 1, our agent will *lose* all but finitely many of the games he plays. Since he loses one dollar per play indefinitely after his final win, his winnings converge to $-\infty$ with probability 1, even though each of his wagers has positive expectation!

Acknowledgement
The first author was partially supported in this research by iARPA.

References

Balduccini, M. (2007). CR-MODELS: An inference engine for CR-Prolog. In C. Baral, G. Brewka, and J. Schlipf (Eds.), *Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07)*, Volume 3662 of *Lecture Notes in Artificial Intelligence*, pp. 18–30. Springer.

Balduccini, M. and M. Gelfond (2003, Mar). Logic Programs with Consistency-Restoring Rules. In P. Doherty, J. McCarthy, and M.-A. Williams (Eds.), *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pp. 9–18.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving with answer sets*. Cambridge University Press.

Baral, C., M. Gelfond, and N. Rushton (2004, Jan). Probabilistic Reasoning with Answer Sets. In *Proceedings of LPNMR-7*.

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

Baral, C. and M. Hunsaker (2007). Using the probabilistic logic programming language p-log for causal and counterfactual reasoning and non-naive conditioning. In *Proceedings of IJCAI-2007*, pp. 243–249.

Gelfond, M. and V. Lifschitz (1988). The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pp. 1070–1080.

Gelfond, M. and V. Lifschitz (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*(3/4), 365–386.

Gelfond, M., N. Rushton, and W. Zhu (2006). Combining logical and probabilistic reasoning. AAAI 2006 Spring Symposium Series, pp. 50–55.

Hayes, P. J. and J. McCarthy (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press.

McCarthy, J. (1999). Elaboration tolerance. In progress.

Pearl, J. (1988). *Probabistic reasoning in intelligent systems: networks of plausable inference*. Morgan Kaufmann.

Pearl, J. (2000). *Causality*. Cambridge University Press.

Pereira, L. M. and C. Ramli (2009). Modelling decision making with probabilistic causation. *Intelligent Decision Technologies (IDT)*. to appear.

Reiter, R. (1978). *On Closed World Data Bases*, pp. 119–140. Logic and Data Bases. Plenum Press.