The Independent Choice Logic and Beyond

David Poole

Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, B.C., Canada V6T 1Z4
poole@cs.ubc.ca
http://www.cs.ubc.ca/spider/poole/

Abstract. The Independent Choice Logic began in the early 90's as a way to combine logic programming and probability into a coherent framework. The idea of the Independent Choice Logic is straightforward: there is a set of independent choices with a probability distribution over each choice, and a logic program that gives the consequences of the choices. There is a measure over possible worlds that is defined by the probabilities of the independent choices, and what is true in each possible world is given by choices made in that world and the logic program. ICL is interesting because it is a simple, natural and expressive representation of rich probabilistic models. This paper gives an overview of the work done over the last decade and half, and points towards the considerable work ahead, particularly in the areas of lifted inference and the problems of existence and identity.

1 Introduction

There are good normative arguments for using logic to represent knowledge [Nilsson, 1991; Poole et al., 1998]. These arguments are usually based on reasoning with symbols with an explicit denotation, allowing relations amongst individuals, and permitting quantification over individuals. This is often translated as needing (at least) the first-order predicate calculus.

There are also good normative reasons for using Bayesian decision theory for decision making under uncertainty [Neumann and Morgenstern, 1953; Savage, 1972]. These arguments can be intuitively interpreted as seeing decision making as a form of gambling, and that probability and utility are the appropriate calculi for gambling. These arguments lead to the assignment of a single probability to a proposition; thus leading to the notion of probability as a measure of subjective belief.

These two normative arguments are not in conflict with each other. Together they suggest having probability measures over rich structures. How this can be done in a simple, straightforward manner is the motivation behind a large body of research over the last 20 years.

The independent choice logic (ICL) started off as Probabilistic Horn Abduction [Poole, 1991a,b, 1993a,b] (the first three of these papers had a slightly different language), which allowed for probabilistically independent choices and a logic program to give the consequences of the choices. The independent choice logic extends probabilistic Horn abduction in allowing for multiple agents each making their own choices

[Poole, 1997b] (where nature is a special agent who makes choices probabilistically) and in allowing negation as failure in the logic [Poole, 2000b].

The ICL is still one of the simplest and most powerful representations available. It is simple to define, straightforward to represent knowledge in and powerful in that it is a Turing-complete language that can represent arbitrary finite[1] probability distributions at least as compactly as can Bayesian belief networks. It can also represent infinite structures such as Markov chains.

In this paper we overview the base logic, and give some representation, inference and learning challenges that still remain.

2 Background

2.1 Logic Programming

The independent choice logic builds on a number of traditions. The first is the idea of logic programs [Lloyd, 1987].

A logic program is built from constants (that denote particular individuals), variables (that are universally quantified over the set of individuals), function symbols (that are used to indirectly describe individuals), predicate symbols (that denote relations). A term is either a constant, a variable or of the form $f(t_1, \ldots, t_k)$ where f is a function symbol and the t_i are terms. A predicate symbol applied to a set of terms is an atomic formula (or just an atom). A **clause** is either an atom or is of the form

$$h \leftarrow a_1 \wedge \cdots \wedge a_k$$

where h is an atom and each a_i is an atom or the negation of an atom. We write the negation of atom a as $\neg a$. A logic program is a set of clauses. We use the Prolog convention of having variables in upper case, and constants, predicate symbols and function symbols in lower case. We also allow for Prolog notation for lists and allow the standard Prolog infix operators.

A ground atom is one that does not contain a variable. The grounding of a program is obtained by replacing the variables in the clauses by the ground terms. Note that if there are function symbols, the grounding contains countably infinitely many clauses.

Logic programs are important because they have:

– a logical interpretation in terms of truth values of clauses (or their completion [Clark, 1978]). A logic program is a logical sentence from which one can ask for logical consequences.
– a procedural semantics (or fixed-point semantics). A logic program is a non-deterministic pattern-matching language where predicate symbols are procedures and function symbols give data structures.

[1] It can represent infinite distributions, just not arbitrarily complex ones. A simple counting argument says that no finite language can represent arbitrary probability distributions over countable structures.

– a database semantics in terms of operations on the relations denoted by predicate symbols. As a database language, logic programs are more general than the relational algebra as logic programs allow recursion and infinite relations using function symbols.

Thus a logic program can be interpreted as logical statements, procedurally or as a database and query language.

Logic programming research has gone in two general directions. In the first, are those frameworks, such as acyclic logic programs [Apt and Bezem, 1991], that ensure there is a single model for any logic program. Acyclic logic programs assume that all recursions for variable-free queries eventually halt. In particular, a program is acyclic if there is assignment of an natural number to each ground atom so that the natural number assigned to the head of each clause in the grounding of the program is greater than the atoms in the body. The acyclicity disallows programs such as $\{a \leftarrow \neg a\}$ and $\{a \leftarrow \neg b, b \leftarrow \neg a\}$. Intuitively, acyclic programs are those that don't send top-down interpreters, like Prolog, into infinite loops. Under this view, cyclic programs are buggy as recursions are not well founded.

The other direction, exemplified by Answer Set Programming [Lifschitz, 2002], allows for cyclic theories, and considers having multiple models as a virtue. These multiple models can correspond to multiple ways the world can be. Baral et al. [2004] have investigated having probability distributions over answer sets.

The stable model semantics [Gelfond and Lifschitz, 1988] provides a semantics for logic programs where the clauses contain negations in the body (i.e., for "negation as failure"). The stable model semantics is particularly simple with acyclic logic programs [Apt and Bezem, 1991]. A model specifies the truth value for each ground atom. A stable model M is a model where a ground atom a is true in M if and only if there is a clause in the grounding of the logic program with a as the head where the body is true in M. An acyclic logic program has a unique stable model.

Clark [1978] gives an alternative definition of negation as failure in terms of completions of logic programs. Under Clark's completion, a logic program means a set of if and only if definitions of predicates. Acyclic logic programs have the nice property that what is true in the unique stable model corresponds to what logically follows from Clark's completion.

2.2 Belief Networks

A belief network or Bayesian network [Pearl, 1988] is a representation of independence amongst a finite set of random variables. In this paper we will write random variables starting with a *lower case* letter, so they are not confused with logical variables.

In particular, a belief network uses an acyclic directed graph to represent the dependence amongst a finite set of random variables: the nodes in the graph are the variables, and the network represents the independence assumption that a node is independent of its non-descendants given its parents. That is, if x is a random variable, and $par(x)$ is the set of parents of in the graph, then $P(x|par(x),u) = P(x|par(x))$. It follows from the chain rule of probability that if $x_1,\ldots,x_n$ are all of the variables, then

$$P(x_1,\ldots,x_n) = \prod_{i=1}^{n} P(x_i|par(x_i))$$

The probability distribution $P(x_1,\ldots,x_n)$ can be used to define any conditional probability.

Bayesian networks have become very popular over the last 20 years, essentially because:

- The independence assumption upon which belief networks are based is useful in practice. In particular, causality would be expected to obey the independence assumption if the direct causes of some event are the parents. The notion of locality of influence is a good modelling assumption for many domains.
- There is a natural specification of the parameters. Humans can understand and debug the structure and the numbers. The specification in terms of conditional probability means that the probabilities can be learned independently; adding and removing variables from the models only have a local effect.
- There are algorithms that can exploit the structure for efficient inference.
- The probabilities and the structure can be learned effectively.

Note that the first two properties are not true of undirected models such as Markov networks (see e.g., Pearl [1988], pages 107–108). Markov Logic Networks [Richardson and Domingos, 2006] inherit all of the problems of undirected models.

Example 1. Consider the problem of diagnosing errors that students make on simple multi-digit addition problems [Brown and Burton, 1978]:

$$\begin{array}{cccc} & & x_2 & x_1 \\ + & & y_2 & y_1 \\ \hline & z_3 & z_2 & z_1 \end{array}$$

The students are presented with the digits x_1, x_2, y_1 and y_2 and are expected to provide the digits z_1, z_2 and z_3. From observing their behaviour, we want to infer whether they understand how to do arithmetic, and if not, what they do not understand so that they can be taught appropriately.

For simplicity, let's assume that the student can make systematic and random errors on the addition of digits and on the carrying. This problem can be represented as a belief network as depicted in Figure 1. The tabular representation of the conditional probabilities often used in belief networks makes the representation cumbersome, but it can be done.

In this figure, x_1 is a random variable with domain $\{0,\ldots,9\}$ that represents the digit in the top-right. Similarly for the variables x_2, y_1, y_2, z_1, z_2, z_3. The $carry_i$ variables represent the carry into digit i, and have domain $\{0,1\}$. The Boolean variable *knows_addition* represents whether the student knows basic addition of digits. (A Boolean variable has domain $\{true, false\}$). The variable *knows_carry* has domain $\{$*knowCarry*, *carryZero*, *carryRandom*$\}$ representing whether they know how to carry, whether they

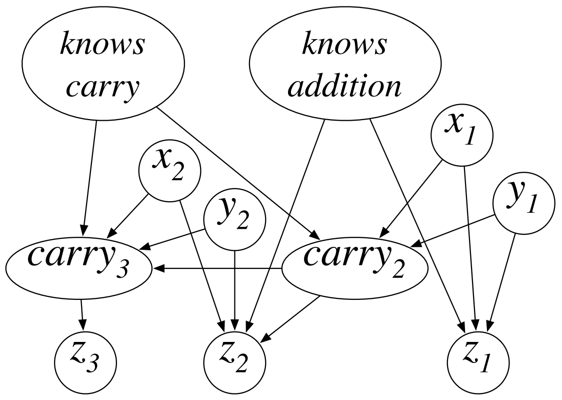

Fig. 1. A belief network for the simple addition example

always carry zero, or whether they carry randomly (i.e., we are modelling more than one error state). By conditioning on the observed values of the x_i, y_i and z_i variables, inference can be used to compute the probability that a student knows how to carry or knows addition.

The belief network representation for the domain of diagnosing students with addition problems becomes impractical when there are multiple problems each with multiple digits, multiple students, who take the same or different problems at different times (and change their skills through time).

One way to represent this is to duplicate nodes for the different digits, problems, students and times. The resulting network can be depicted using plates [Buntine, 1994], as in Figure 2. The way to view this representation is that there are copies of the variables

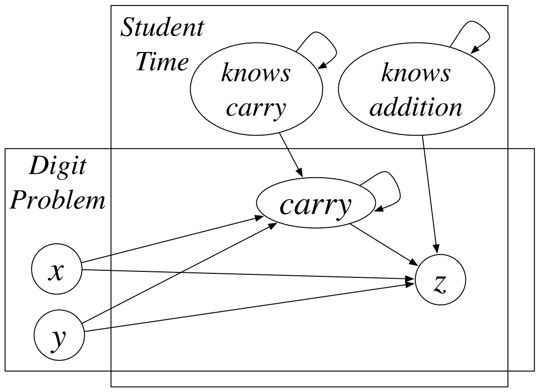

Fig. 2. A belief network with plates addition example

in the Student/Time plate for each student-time pair. There are copies of the variables in the Digit/Problem plate for each digit-problem pair. Think of the plates as coming out of the diagram. Thus, there are different instances of variables *x* and *y* for each digit and each problem; different instances of *carry* and *z* for each digit, problem, student and time; and different instances of *knowsCarry* and *knowsAddition* for each student and time. Note that there can be cycles in this graph when the dependence is on different instances of variables. For example, there is a cycle on the node *carry* as the carry for one digit depends on the carry from the previous digit. The cycle on *knowsCarry* is because a student's knowledge of carrying depends on the student's knowledge at the previous time.

The plate notation is very convenient and natural for many problems and leads to what could be called parametrized belief networks that are networks that are built from templates [Horsch and Poole, 1990]. Note that it is difficult to use plates when one variable depends on different instances of the same relation. For example, if whether two authors collaborate depends on whether they have coauthored a paper (collaborate depends on two different instances of an author relationship). Heckerman et al. [2004] show how plates relate to PRMs and discuss limitations of both models.

2.3 The Independent Choice Logic

The ICL can either be seen as adding independent stochastic inputs to a logic program, or as a way to have rule-based specification of Bayesian networks with logical variables (or plates).

We assume that we have atomic formulae as logic programming. We allow function symbols.

An **atomic choice** is an atom that does not unify with the head of any clause. An **alternative** is a set of atomic choices. A **choice space** is a set of alternatives such that the atomic choices in different alternatives do not unify.

An ICL theory consists of:

F the facts, an acyclic logic program

C a choice space, which is a set of sets of atoms. The elements of the choice space are called alternatives. The elements of the alternatives are called atomic choices. Atomic choices in the same or different alternatives cannot unify with each other. Atomic choices cannot unify with the head of any clause in *F*.

P_0 a probability distribution over the alternatives in *C*. That is $P_0 : \cup\mathbf{C} \to [0,1]$ such that

$$\forall \chi \in \mathbf{C} \sum_{\alpha \in \chi} P_0(\alpha) = 1$$

The restrictions on the unification of atomic choices are there to enable a free choice of an atomic choice from each alternative.

Example 2. Here is a meaningless example (that is simple enough to show the semantics below):

$$\mathbf{C} = \{\{c_1, c_2, c_3\}, \{b_1, b_2\}\}$$

$$
\begin{aligned}
\mathbf{F} = \{ & f \leftarrow c_1 \wedge b_1, f \leftarrow c_3 \wedge b_2, \\
& d \leftarrow c_1, \qquad d \leftarrow \neg c_2 \wedge b_1, \\
& e \leftarrow f, \qquad e \leftarrow \neg d \}
\end{aligned}
$$

$P_0(c_1) = 0.5$ $P_0(c_2) = 0.3$ $P_0(c_3) = 0.2$
$P_0(b_1) = 0.9$ $P_0(b_2) = 0.1$

2.4 Semantics

The semantics is defined in terms of possible worlds.

A **total choice** for choice space C is a selection of exactly one atomic choice from each grounding of each alternative in C. Note that while choice spaces can contain free variables, a total choice does not.

There is a **possible world** for each total choice. What is **true** in a possible world is defined by the atoms chosen by the total choice together with the logic program. In particular an atom is true in a possible world if it is in the (unique) stable model of the total choice corresponding to the possible world together with the logic program [Poole, 2000b]. The acyclicity of the logic program and the restriction that atomic choices don't unify with the head of clauses guarantees that there is a single model for each possible world.

For the case where there are only a finite number of possible worlds (when there are no function symbols), the probability for a possible world is the product of the probabilities of the atomic choices that make up the possible world. The probability of any proposition is the sum of the probabilities of the possible worlds in which the proposition is true.

Example 3. In the ICL theory of example 2, there are six possible worlds, and each possible world can be given a probability:

$$
\begin{array}{lllll}
w_1 \models c_1\ b_1\ \ f\ \ d\ \ e & P(w_1) = 0.45 \\
w_2 \models c_2\ b_1\ \neg f\ \neg d\ \ e & P(w_2) = 0.27 \\
w_3 \models c_3\ b_1\ \neg f\ \ d\ \neg e & P(w_3) = 0.18 \\
w_4 \models c_1\ b_2\ \neg f\ \ d\ \neg e & P(w_4) = 0.05 \\
w_5 \models c_2\ b_2\ \neg f\ \neg d\ \ e & P(w_5) = 0.03 \\
w_6 \models c_3\ b_2\ \ f\ \neg d\ \ e & P(w_6) = 0.02
\end{array}
$$

The probability of any proposition can be computed by summing the measures of the worlds in which the proposition is true. For example

$$P(e) = \mu(\{w_1, w_2, w_5, w_6\}) = 0.45 + 0.27 + 0.03 + 0.02 = 0.77$$

where μ is the measure on sets of possible worlds.

When there are function symbols, there are infinitely many possible worlds, and there needs to be a more sophisticated definition of probabilities. To define the probabilities, we put a measure over sets of possible worlds. In particular, we define the sigma algebra of sets of possible worlds that can be described by finite logical formulae of ground atomic choices. We define a probability measure over this algebra as follows.

Any formula of ground atomic choices can be put into disjunctive normal form, such that the disjuncts are mutually exclusive. That is, it can be put in the form

$$(a_{11} \wedge \ldots \wedge a_{1k_1}) \vee \ldots \vee (a_{m1} \wedge \ldots \wedge a_{mk_m})$$

such that for each $i \neq j$ there is some such s, t that a_{is} and a_{jt} are in the grounding of an alternative. The set of all possible worlds where this formula is true, has measure

$$\sum_i \prod_j P_0(a_{ij})$$

Note that this is assuming that different ground instances of alternatives are probabilistically independent.

The probability of a proposition is the measure of the set of possible worlds in which the proposition is true.

2.5 ICL and Belief networks

It may seem that, with independent alternatives, that the ICL is restricted in what it can represent. This is not the case; the ICL can represent anything that is representable by a belief network. Moreover the translation is local, and there is the same number of alternatives as there are free parameters in the belief network.

A random variable having a value is a proposition and is represented in ICL as a logical atom. Random variable x having value v can be represented as the atom $x(v)$, with the intended interpretation of $x(v)$ being that x has value v. We can also use the infix relation $=$ and write $x = v$. Note that there is nothing special about $=$ (the only thing built-in is that it is an infix operator). As long as you use a consistent notation for each random variable, any syntax can be used, except that names that start with an upper case represent logical variables (following the Prolog tradition), not random variables.

A Boolean random variable x can be optionally directly represented as an ICL atom x; i.e., $x = true$ is x and $x = false$ is $\neg x$.

Conditional probabilities can be locally translated into rules, as in the following example.

Example 4. Suppose a, b and c are variables, where b and c are the parents of a. We could write the conditional probability as a single rule:

$$a(A) \leftarrow b(B) \wedge c(C) \wedge na(A,B,C)$$

where there is an alternative for $na(A,B,C)$ for each value of B and C. For example, if a, b and c were Boolean (have domain $\{true, false\}$), one alternative is:

$$\{na(true, true, false), na(false, true, false)\}$$

where $P_0(ca(true, true, false))$ has the same value as $P(a|b, \neg c)$ in the belief network.

As a syntactic variant, if you had decided to use the $=$ notation, this clause could be written as:

$$a = A \leftarrow b = B \wedge c = C \wedge aifbnc(A,B,C)$$

If a, b and c are Boolean variables, replacing the $P(a|b,\neg c)$ in the belief network, we could have rules such as

$$a \leftarrow b \wedge \neg c \wedge \mathit{aifbnc}$$

where *aifbnc* is an atomic choice where $P_0(\mathit{aifbnc})$ has the same value as the conditional probability as $P(a|b,\neg c)$ in the belief network.

As a syntactic variant, this can be abbreviated as

$$a \leftarrow b \wedge \neg c : p_0$$

where p_0 is $P(a|b,\neg c)$. In general

$$H \leftarrow b : p$$

is an abbreviation for $H \leftarrow b \wedge n$, where n is an atom that contains a new predicate symbol and contains all of the free variables of the clause. This notation, however, tends to confuse people (students who are familiar with logic programming), and they make more mistakes than when they have a clear separation of the logical and probabilistic parts. It is not as useful when there are logical variables, as there tends to be many numbers (as in the table for $\mathit{aifbnc}(A,B,C)$), often we don't want a fully parametrized atomic choice, and you often want to say what happens when the atomic choice is false.

The ICL representation lets us naturally specify context-specific independence [Boutilier et al., 1996; Poole, 1997a], where, for example, a may be independent of c when b is false but be dependent when b is true. Context-specific independence is often specified in terms of a tree for each variable; the tree has probabilities at the leaves and parents of the variable on the internal nodes. It is straightforward to translate these into the ICL. The logic program also naturally represents the "noisy or", when the bodies are not disjoint which is a form of causal independence [Zhang and Poole, 1996]. Standard algorithms such as clique-tree propagation are not good at reasoning with these representations, but there are ways to exploit noisy-or and context specific independence using modifications of variable elimination [Zhang and Poole, 1996; Díez and Galán, 2002; Poole and Zhang, 2003] or recursive conditioning [Allen and Darwiche, 2003].

Example 5. Continuing our addition example, it is difficult to specify a Bayesian network even for the simple case of adding two digits case as shown in Figure 1, as specifying how, say z_2 depends on its parents is non-trivial, let alone for arbitrary digits, problems, students and times.

In this example, we will show the complete axiomatization for z that runs in our CILog2 interpreter[2]. The plates of Figure 2 correspond to logical variables. For example, $z(D,P,S,T)$ gives the z value for the digit D, problem P, student S, at time T. $x(D,P)$ is the x value for digit D and problem P. $\mathit{knowsCarry}(S,T)$ is true when student S knows how to carry at time T.

The rules for z are as follows. Note that the symbol "=" here is just a syntactic sugar; it just as easily could be made the last argument of the predicate.

[2] The complete code can be found at http://www.cs.ubc.ca/spider/poole/ci2/code/cilog/CILog2.html

If the student knows addition and didn't make a mistake on this case, they get the right answer:

$$\begin{aligned}
&z(D,P,S,T) = V \leftarrow \\
&\quad x(D,P) = Vx \wedge \\
&\quad y(D,P) = Vy \wedge \\
&\quad carry(D,P,S,T) = Vc \wedge \\
&\quad knowsAddition(S,T) \wedge \\
&\quad \neg mistake(D,P,S,T) \wedge \\
&\quad V \text{ is } (Vx + Vy + Vc) \text{ div } 10.
\end{aligned}$$

If the student knows addition but made a mistake, they pick a digit at random:

$$\begin{aligned}
&z(D,P,S,T) = V \leftarrow \\
&\quad knowsAddition(S,T) \wedge \\
&\quad mistake(D,P,S,T) \wedge \\
&\quad selectDig(D,P,S,T) = V.
\end{aligned}$$

If the student doesn't know addition, they pick a digit at random:

$$\begin{aligned}
&z(D,P,S,T) = V \leftarrow \\
&\quad \neg knowsAddition(S,T) \wedge \\
&\quad selectDig(D,P,S,T) = V.
\end{aligned}$$

The alternatives are:

$$\begin{aligned}
&\{noMistake(D,P,S,T), mistake(D,P,S,T)\} \\
&\{selectDig(D,P,S,T) = V \mid V \in \{0..9\}\}
\end{aligned}$$

There are similar rules for $carry(D,P,S,T)$ that depend on $x(D,P)$, $y(D,P)$, $knowsCarry(S,T)$ and $carry(D1,P,S,T)$ where $D1$ is the previous digit. And similar rules for $knowsAddition(S,T)$ that depends on the previous time.

By observing x, y and z for a student on various problems, we can query on the probability the student knows addition and knows how to carry.

2.6 Unknown Objects

BLOG [Milch et al., 2005] claims to deal with unknown objects. In this section we will show how to write one of the BLOG example in ICL. First note that BLOG has many built-in procedures, and ICL (as presented) has none. I will simplify the example only to make concrete the distributions used which are not specified in the BLOG papers.

I will do the aircraft example, which is Example 3 of Milch et al. [2005]. As I will make the low levels explicit I will assume that the locations are a 10x10 grid and there are 8 directions of aircraft.

First we can generate a geometric distribution of the number of aircraft[3]. We can do this by generating the number by repeatedly asking whether there are any more; the resulting number follows a geometric distribution with parameter P (which must be specified when *numObj* is called). $\mathit{numObj}(T, N, N1, P)$ is true if there are $N1$ objects of type T given there are N. Note that the type is provided because we want the number of planes to be independent of the number of other objects (even though it isn't strictly needed for this example):

$\mathit{numObj}(T, N, N, P) \leftarrow$
  $\neg \mathit{more}(T, N, P).$
$\mathit{numObj}(T, N, N2, P) \leftarrow$
  $\mathit{more}(T, N, P) \wedge$
  $N1$ is $N + 1 \wedge$
  $\mathit{numObj}(T, N + 1, N2, P).$

Where the alternative is (in CILog syntax):

prob $\mathit{more}(T, N, P) : P.$

which means that $\{\mathit{more}(T, N, P), \neg \mathit{more}(T, N, P)\}$ is an alternative with $P(\mathit{more}(T, N, P)) = P$. Note that we could have equivalently used $\mathit{noMore}(T, N, P)$ instead of $\neg \mathit{more}(T, N, P)$.

Note also that if the probability changes with the number of objects (as in a Poisson distribution), that can be easily represented too.

We can define the number of aircraft using *numObj* with $P = 0.8$:

$\mathit{numAircraft}(N) \leftarrow$
  $\mathit{numObj}(\mathit{aircraft}, 0, N, 0.8).$

The aircraft will just be described in terms of the index from 1 to the number of aircraft:

$\mathit{aircraft}(I) \leftarrow$
  $\mathit{numAircraft}(N) \wedge$
  $\mathit{between}(1, N, I).$

where $\mathit{between}(L, H, V)$ is true if $L \leq V \leq H$ and is written just as it would be in Prolog. Note that $\mathit{aircraft}(I)$ is true in the worlds where there are at least I aircraft.

We can now define the state of the aircraft. The state of the aircraft with consist of an x-coordinate (from 0 to 9), a y-coordinate (from 0 to 9), a direction (one of the 8 directions) and a predicate to say whether the aircraft is inside the grid. Let $\mathit{xpos}(I, T, V)$ mean the x-position of aircraft I at time T is V if it is in the grid, and the value V can be arbitrary if the aircraft I is outside the grid.

[3] I would prefer to have the initial number of aircraft and to allow for new aircraft to enter, but as the BLOG paper did not do this, I will not either. I will assume that all of the aircraft are in the grid at the start, these can leave, and no new aircraft can arrive.

We will assume the initial states are independent, so they can be stated just in choices. One alternative defines the initial x-coordinate.

$$\{xpos(I,0,0), xpos(I,0,1), \ldots, xpos(I,0,9)\}$$

with $P_0(xpos(I,0,V)) = 0.1$.

We can axiomatize the dynamics similarly to the examples of Poole [1997b], with rules such as:

$xpos(I, next(T), P) \leftarrow$
$\quad xpos(I, T, PrevPos) \wedge$
$\quad direction(I, T, Dir) \wedge$
$\quad xDer(I, T, Dir, Deriv) \wedge$
$\quad P$ is $PrevPos + Deriv$.

Where $xDer(I,T,Dir,Deriv)$ gives the change in the x-position depending on the direction. It can be defined using CILog syntax using alternatives such as:

prob $xDer(I,T,east,0) : 0.2, xDer(I,T,east,1) : 0.7, xDer(I,T,east,2) : 0.1$

We can now define how blips occur. Let $blip(X,Y,T)$ be true if a blip occurs at position X-Y at time T. Blips can occur at random or because of aircraft. We could either have a distribution over the number of blips, as we did for aircraft, and have a random variable for the location of each blip. Alternatively, we could have a random variable for the existence of a blip at each location. To do the latter, we would have:

$blip(X,Y,T) \leftarrow$
$\quad blipRandomlyOccurs(X,Y,T)$.

Suppose that blips can randomly occur at any position with probability 0.02, independently for each position and time. This can be stated as:

prob $blipRandomlyOccurs(X,Y,T) : 0.1$.

To axiomatize how blips can be produced by aircraft we can write[4]:

$blip(X,Y,T) \leftarrow$
$\quad aircraft(I) \wedge$
$\quad inGrid(I,T) \wedge$
$\quad xpos(I,T,X) \wedge$
$\quad ypos(I,T,Y) \wedge$
$\quad producesBlip(I,T)$.

Aircraft produce blips where they are with probability 0.9 can be stated as:

prob $producesBlip(I,T) : 0.1$.

[4] I will leave it as an exercise to the reader to show how to represent the case where there can be noise in the location of the blip.

Observing blips over time, you can ask the posterior distributions of the number of aircraft, their positions, etc. [Poole, 1997b] gives some similar examples (but without uncertainty over the number).

Ultimately, you want to axiomatize the actual dynamics of blip production; how are blips actually produced? Presumably the experts in radars have a good intuition (which can presumably be combined with data).

The complete axiomatization is available at the CILog2 web site (Footnote 2). Note that the CILog implementation can solve this as it generates only some of the proofs and bounds the error [Poole, 1993a] (although not as efficiently as possible; see Section 3).

BLOG lets you build libraries of distributions in Java. ICL, lets you build them in (pure) Prolog. The main difference is that a pure Prolog program is an ICL program (and so the libraries will be in ICL), but a Java program is not a BLOG program.

2.7 ICL as a Programming Language

The procedural interpretation of logic programs gives another way to look at ICL. It turns out that any belief network can be represented as a deterministic system with (independent) probabilistic exogenous inputs [Pearl, 2000, p. 30]. One technique for making a probabilistic programming language is to use a standard programming language to define the deterministic system and to allow for random inputs. This is the basis for a number of languages which differ in the language used to specify the deterministic system:

- ICL uses acyclic logic programs (they can even have negation as failure) to specify the deterministic system
- IBAL [Pfeffer, 2001] uses an ML-like functional language to specify the deterministic system
- A-Lisp [Andre and Russell, 2002] uses Lisp to specify the deterministic system
- CES [Thrun, 2000] uses C++ to specify the deterministic system.

While each of these have their advantages, the main advantage if ICL is the declarative semantics and the relational view (it is also an extension of Datalog).

All of these can be implemented stochastically, where the inputs are chosen using a random number generator and the programming language then gives the consequences. You can do rejection sampling by just running the program in the randomly generated inputs, but the real challenge is to do more sophisticated inference which has been pursued by for all of these.

2.8 ICL, Abduction and Logical Argumentation

Abduction is a powerful reasoning framework. The basic idea of abduction is to make assumptions to explain observations. This is usually carried out by collecting the assumptions needed in a proof and ensuring they are consistent.

The ICL can also be seen as a language for abduction. In particular, if all of the atomic choices are assumable (they are abducibles or possible hypotheses). An **explanation**[5] for g is a consistent set of assumables that implies g. A set of atomic choices is consistent if there is at most one element in any alternative.

Recall that the semantic of ICL is defined in terms of a measure over set of atomic choices. Abduction can be used to derive those atomic choices over which the measure is defined.

Each of the explanations has an associated probability obtained by computing the product of the probabilities of the atomic choices that make up the explanation. If the rules for each atom are mutually exclusive, the probability of g can be computed by summing the probabilities of the explanations for g [Poole, 1993b, 2000b]. We need to do something a bit more sophisticated if the rules are not disjoint or contain negation as failure [Poole, 2000b]. In these cases we can make the set of explanations disjoint (in a similar way to build a decision tree from rules) or find the duals of the set of explanations of g to find the explanations of $\neg g$.

If we want to do evidential reasoning and observe *obs*, we compute

$$P(g|obs) = \frac{P(g \wedge obs)}{P(obs)}$$

In terms of explanations, we can first find the explanations for *obs* (which would give us $P(obs)$) and then try to extend these explanations to also explain g (this will give us $P(g \wedge obs)$). Intuitively, we explain all of the observations and see what these explanations also predict. This is similar to proposals in the non-monotonic reasoning community to mix abduction and default reasoning [Poole, 1989; Shanahan, 1989; Poole, 1990].

We can also bound the prior and posterior probabilities by generating only a few of the most plausible explanations (either top-down [Poole, 1993a] or bottom-up [Poole, 1996]). Thus we can use inference to find the best explanations to do sound (approximate) probabilistic reasoning.

2.9 Other Formalisms

There are other formalisms that combine logic programming and probabilities.

Some of them such as Bayesian Logic Programs [Kersting and De Raedt, 2007] use some of the theory of logic programming, but mean different things by their logic programs. For example, the *allHappy* program (where $allHappy(L)$ is true if all elements of list L are happy):

$$allHappy([]).$$
$$allHappy([X|A]) \leftarrow happy(X) \wedge allHappy(A).$$

is a perfectly legal ICL program and means exactly the same as it does in Prolog. The meaning does not depend on whether there is uncertainty about whether someone is

[5] We need to extend the definition of explanation to account for negation as failure. The explanation of $\neg a$ are the duals of the explanations of a [Poole, 2000b].

happy, or even uncertainty in the elements of the list. However, this is not a Bayesian logic program (even if *allHappy* was a Boolean random variable).

Bayesian logic program do not mean the same things by atoms in logic programs. In normal logic programs, atoms represent propositions (and so can be combined with logical expressions). In Bayesian logic programs they mean random variables. The rules in ICL mean implication, whereas in BLP, the rules mean probabilistic dependency. BLP does not allow negation as failure, but ICL does.

In ICL, familiar theorems of logic programs are also theorems of ICL. Any theorem about acyclic logic programs is also a theorem of ICL (as it is true in all possible worlds). In particular, Clark's completion [Clark, 1978] is a theorem of ICL. For example, the completion of *allHappy* is true in the ICL when the above definition of *allHappy* is given in the facts. This is true even if $happy(X)$ is stochastic: $happy(a)$ may have different values in different possible worlds, and so might $allHappy([a,b,c])$, but the completion is true in all of the possible worlds.

Kersting and De Raedt [2007] claim that "Bayesian logic programs [compared to ICL theories] have not as many constraints on the representation language, ..., have a richer representation language and their independence assumptions represent the causality of the domain". All these claims are false. ICL can represent arbitrary logic programs with no restrictions except the acyclicity restriction that recursions have to be well-founded (none of the textbooks on logic programming I know of contains any logic programs that are not acyclic, although they do have logic programs that contains cuts and input/output that ICL does not handle.). ICL is Turing-complete. Pearl [2000] *defines* causality in terms of structural equation models with exogenous noise. The ICL represents this causality directly with the structural equation models represented as logic programs.

Note that the logical reasoning in Bayesian logic programming constructs a belief network, whereas logical reasoning in ICL *is* probabilistic reasoning. Finding the proofs of the observation and query is enough to determine the posterior probability of the query [Poole, 1993a]. Reasoning in ICL has inspired many of the algorithms for Bayesian network inference [Zhang and Poole, 1996; Poole, 1996; Poole and Zhang, 2003]. We should not assume that we can just pass of the probabilistic inference problem to a general-purpose solver and expect it to work (although [Chavira et al., 2006] comes close to this goal) as there is much more structure in the high-level representations.

Stochastic logic programs [Muggleton, 1996] are quite different in their goal to the other frameworks presented here. Stochastic logic programs give probability distributions over proofs, rather than defining the probability that some proposition is true.

3 Ongoing Research

This section outlines some challenges that we are currently working on. These correspond to representation, inference and learning.

3.1 Continuous Variables

There is nothing in the ICL semantics that precludes having continuous random variables, or even continuously many random variables. In the example of aircraft and blips, we could have a continuous random variable that is the position of the aircraft. We could also have continuously many random variables, about whether there is a blip at each of continuously many x-y positions.

The semantics of ICL mimics the standard definition of the limiting process of a continuous variable. The logical formulae could describe finer partitions of the continuous space, and we get the standard definition.

However, reasoning in terms of finer partitions is not computationally satisfactory. It is better to define continuous variables in terms of a mixture of kernel functions, such as mixtures of Gaussian distributions, truncated Gaussians [Cozman and Krotkov, 1994] or truncated exponential distributions [Cobba and Shenoy, 2006]. This can be done by having Gaussian alternatives. Allowing Gaussian alternatives and conditions in the logic programs, means that the program has to deal with truncated Gaussians; but it also means that it is easy to represent truncated Gaussians in terms of logic programs with Gaussian alternatives.

There are still many problems to be solved to get this to work satisfactorily. It needs to be expressive enough to state what we need, but it also needs to be able to be reasoned with efficiently.

3.2 Reasoning in the ICL

To reason in the ICL we can either do

– variable elimination (marginalization) to simplify the model [Poole, 1997a]. We sum out variables to reduce the detail of the representation. This is similar to partial evaluation in logic programs.
– Generating some of the explanations to bound the probabilities [Poole, 1993a, 1996]. If we generated all of the explanations we could compute the probabilities exactly, but there are combinatorially many explanations. It should be possible to combine this with recursive conditioning [Darwiche, 2001] to get the best of both worlds.
– Stochastic simulation; generating the needed atomic choices stochastically, and estimating the probabilities by counting the resulting proportions.

One of the things that we would like in a logical language like the ICL to allow lifted inference, where we don't ground out the theory, but reason at the first-order level, with unification. There have been a number of attempts at doing this for various simple languages [Poole, 1997a, 2003; de Salvo Braz et al., 2005], but the final solution remains elusive. The general idea of [Poole, 2003] is that we can do lifted reasoning as in theorem proving or as in Prolog, using unification for matching, but instead of applying a substitution such as $\{X/c\}$, we need to split on $X = c$, giving the equality case, $X = c$, and the inequality case, $X \neq c$. Lifted inference gets complicated when we have aggregation, such as the "or" in the logic program, as shown in the following example:

Example 6. Consider the simple ICL theory with a single clause

$$f \leftarrow e(Y).$$

where the alternative is $\{e(Y), not\_e(Y)\}$, where $P(e(X)) = \alpha$. In this case, the probability of f depends on the number of individuals. In particular,

$$P(f) = 1 - (1 - \alpha)^n$$

where n is the population size (the number of individuals). It is of this form as we assume that the choices are independent. The probability of f can be computed in $O(\log n)$ time. The challenge is to define a general algorithm to compute probabilities in time less than linear in the population size.

This example has shown why we need to type existential variables, and a population size for each type.

There are more complicated cases where how to solve it in time less than linear in the population size is part of ongoing research:

Example 7. Consider the rules:

$$\begin{aligned}
&f \leftarrow e(Y).\\
&e(Y) \leftarrow d(Y) \wedge n_1(Y).\\
&e(Y) \leftarrow \neg d(Y) \wedge n_2(Y).\\
&d(Y) \leftarrow c(X, Y).\\
&c(X, Y) \leftarrow b(X) \wedge n_3(X, Y).\\
&c(X, Y) \leftarrow \neg b(X) \wedge n_4(X, Y).\\
&b(X) \leftarrow a \wedge n_5(X)\\
&b(X) \leftarrow \neg a \wedge n_6(X)\\
&a \leftarrow n_7
\end{aligned}$$

Where the n_i are atomic choices. Suppose $P(n_i(\cdot)) = \alpha_i$. There are thus, 7 numbers to specify, but the interdependence is very complicated. Suppose X has domain size n and the Y has domain size m, and the grounding of e is $e_1, \ldots, e_m$, and similarly for the other variables. The grounding can be seen as the belief network of Figure 3.

Note that the parents of f are all interdependent. Not only do they depend on a, but on each of the b variables. It is still an open problem to be able to solve networks such as this in time that is less than linear in n and m.

3.3 ICL and Learning

There is a large body of work on learning and belief networks. This means either:

– Using the belief network as a representation for the problem of Bayesian learning of models [Buntine, 1994]. In Bayesian learning, we want the posterior distribution of hypotheses (models) given the data. To handle multiple cases, Buntine uses the notion of plates that corresponds to the use of logical variables in the ICL [Poole, 2000a]. Poole [2000a] shows the tight integration of abduction and induction. These papers use belief networks and the ICL to learn various representations including decision trees and neural networks, as well us unsupervised learning.

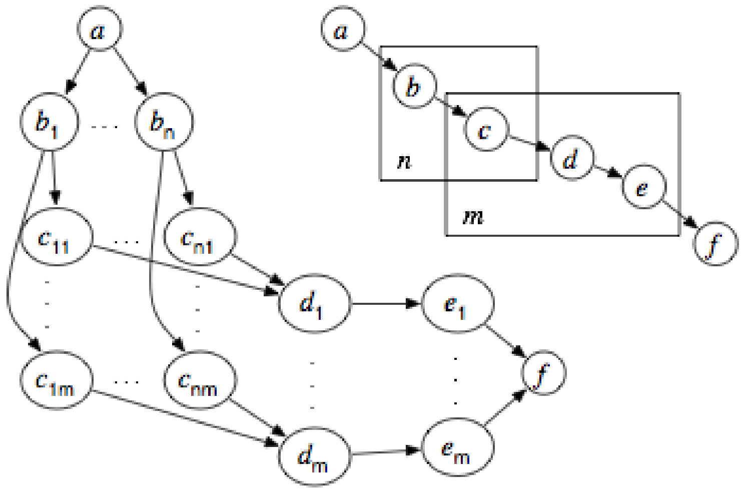

Fig. 3. A belief network and plate representation from Example 7

- Learning the structure and probabilities of belief networks [Heckerman, 1995]. There has been much work on learning parameters for the related system called PRISM [Sato and Kameya, 2001].

There are a number of reasons that the ICL makes a good target language for learning:

- Being based on logic programming, it can build on the successes of inductive logic programming [Muggleton and De Raedt, 1994; Quinlan and Cameron-Jones, 1995; Muggleton, 1995]. The fact that parts of ICL theories are logic programs should aid in this effort.
- There is much local structure that naturally can be expressed in the ICL that can be exploited. One of the most successful methods for learning Bayesian networks is to learn a decision tree for each variable given its predecessors in a total ordering [Friedman and Goldszmidt, 1996; Chickering et al., 1997]. These decision trees correspond to a particular form of ICL rules.
- Unlike many representations such as Probabilistic Relational Models [Getoor et al., 2001], the ICL is not restricted to a fixed number of parameters to learn; it is possible to have representations where each individual has associated parameters. This should allow for richer representations and so better fit to the data. That is, it lets you learn about particular individuals.

3.4 Existence and Identity

The first-order aspects of theorem proving and logic programming are based on Skolemization and Herbrand's theorem. See for example [Chang and Lee, 1973].

Skolemization is giving a name to something that is said to exist. For example, consider the formula $\forall X \exists Y p(X,Y)$. When Skolemizing, we name the Y that exists for each X, say $f(X)$. The Skolemized form of this formula is then $p(X,f(X))$. After Skolemization, there are only universally quantified variables.

Herbrand's theorem [1930] says:

- If a logical theory has a model it has a model where the domain is made of ground terms, and each term denotes itself.
- If a logical theory T is unsatisfiable, there is a finite set of ground instances of formulae of T which is unsatisfiable.

This theorem is the basis for the treatment of variables in theorem proving and logic programming. We may as well reason in terms of the grounding. It also implies that if a logical theory does not contain equality as a built-on predicate, that we can reason as though different ground terms denote different objects.

As soon as we have negation as failure, the implicit assumption that ground terms denote different objects needs to be made explicit.

Languages such as the ICL make two very strong assumptions:

- you know all of the individuals in the domain
- different constants denote different individuals

The first assumption isn't a problem in logic programming and theorem proving, due to Herbrand's theorem. The second assumption, the unique names assumption, is very common in logic programming. Lifting the second assumption is important when we want to reason about identity uncertainty.

To consider the first assumption, suppose that you want to have a naive Bayesian model of what apartment a person likes. Suppose you want to say that if a person likes an apartment, then it's very likely there exists a bedroom in the apartment that is large and green. To do this, it is common to create a constant, say c, for the room that exists. There are two main problems with this. First is semantic, you need to reason about the existence of the room. A common solution is to have existence as a predicate [Pasula et al., 2003; Laskey and da Costa, 2005]. Unfortunately this doesn't work in this situation, as it has to be clear exactly what doesn't exist when the predicate is false, and the properties of the room do not make sense when this room doesn't exist. The second reason is more pragmatic: it isn't obvious how to condition on observations, as you may not know which room that you have observed corresponds to the room c. For example, consider the case where you have observed a green bedroom, but you haven't observed its size, and a large bedroom, but you haven't observed its colour. It isn't well defined (or obvious) how to condition on the observation of c. A solution proposed in [Poole, 2007] is to only have probabilities over well-defined propositions, and for the theory to only refer to closed formulae; this avoids the need to do correspondence between objects in the model and individuals in the world when conditioning. We are currently integrating this idea with the independence assumptions of the ICL.

4 Conclusion

This paper has provided a brief overview of the Independent Choice Logic as well as some issues that arise from representing and reasoning about first-order probabilistic theories. There are some fundamental open issues that cut across representations that we have touched on this paper. Which representations will prove to be most effective remains to be seen.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada Operating Grant OGPOO44121. Thanks to Michael Chiang, Mark Crowley, Jacek Kisyński, Brian Milch, Kristian Kersting and the anonymous reviewers for valuable feedback.

Bibliography

D. Allen and A. Darwiche. New advances in inference by recursive conditioning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 2–10, San Francisco, 2003. Morgan Kaufmann Publishers.

D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *Proc. AAAI-02*, 2002.

K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3-4): 335–363, 1991.

C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. In *Proceedings of LPNMR7*, pages 21–33, 2004.

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In E. Horvitz and F. Jensen, editors, *UAI-96*, pages 115–123, Portland, OR, 1996.

J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155–191, 1978.

W. L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

C. L. Chang and R. C. T. Lee. *Symbolic Logical and Mechanical Theorem Proving*. Academic Press, New York, 1973.

M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning (IJAR)*, 42:4–20, May 2006.

D. M. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *UAI-97*, pages 80–89, 1997.

K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

B. R. Cobba and P. P. Shenoy. Inference in hybrid bayesian networks with mixtures of truncated exponentials. *International Journal of Approximate Reasoning*, 41(3): 257–286, April 2006.

F. Cozman and E. Krotkov. Truncated gaussians as tolerance sets. Technical Report CMU-RI-TR-94-35, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, September 1994.

A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *IJCAI-05*, Edinburgh, 2005. URL http://www.cs.uiuc.edu/ eyal/papers/fopl-res-ijcai05.pdf.

F. J. Díez and S. F. Galán. Efficient computation for the noisy max. *International Journal of Intelligent Systems*, page to appear, 2002.

N. Friedman and M. Goldszmidt. Learning Bayesian networks with local structure. In *UAI-96*, pages 252–262, 1996. URL http://www2.sis.pitt.edu/ dsl/UAI/UAI96/Friedman1.UAI96.html.

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080, Cambridge, MA, 1988.

L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 307–337. Springer-Verlag, 2001.

D. Heckerman. A tutorial on learning with Bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, Mar. 1995. URL http://www.research.microsoft.com/research/dtg/heckerma/heckerma.html. (Revised November 1996).

D. Heckerman, C. Meek, and D. Koller. Probabilistic models for relational data. Technical Report MSR-TR-2004-30, Microsoft Research, March 2004.

M. Horsch and D. Poole. A dynamic approach to probabilistic inference using Bayesian networks. In *Proc. Sixth Conference on Uncertainty in AI*, pages 155–161, Boston, July 1990.

K. Kersting and L. De Raedt. Bayesian logic programming: Theory and tool. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

K. B. Laskey and P. G. C. da Costa. Of klingons and starships: Bayesian logic for the 23rd century. In *Uncertainty in Artificial Intelligence: Proceedings of the Twenty-First Conference*, 2005.

V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138 (1–2):39–54, 2002.

J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation Series. Springer-Verlag, Berlin, second edition, 1987.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *IJCAI-05*, Edinburgh, 2005.

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

S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3,4): 245–286, 1995.

S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

J. V. Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, third edition, 1953.

N. J. Nilsson. Logic and artificial intelligence. *Artificial Intelligence*, 47:31–56, 1991.

H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *NIPS*, volume 15, 2003.

J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2000.

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

A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI-01*, 2001. URL http://www.eecs.harvard.edu/ avi/Papers/ibal.ijcai01.ps.

D. Poole. Logical generative models for probabilistic reasoning about existence, roles and identity. In *22nd AAAI Conference on AI (AAAI-07)*, 2007.

D. Poole. Learning, Bayesian probability, graphical models, and abduction. In P. Flach and A. Kakas, editors, *Abduction and Induction: essays on their relation and integration*. Kluwer, 2000a.

D. Poole. First-order probabilistic inference. In *Proc. Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 985–991, Acapulco, Mexico, 2003.

D. Poole. Explanation and prediction: An architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.

D. Poole. A methodology for using a default and abductive reasoning system. *International Journal of Intelligent Systems*, 5(5):521–548, Dec. 1990.

D. Poole. Representing diagnostic knowledge for probabilistic Horn abduction. In *IJCAI-91*, pages 1129–1135, Sydney, Aug. 1991a.

D. Poole. Representing Bayesian networks within probabilistic Horn abduction. In *UAI-91*, pages 271–278, Los Angeles, July 1991b.

D. Poole. Logic programming, abduction and probability: A top-down anytime algorithm for computing prior and posterior probabilities. *New Generation Computing*, 11(3–4):377–400, 1993a.

D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993b.

D. Poole. Probabilistic conflicts in a search algorithm for estimating posterior probabilities in Bayesian networks. *Artificial Intelligence*, 88:69–100, 1996.

D. Poole. Probabilistic partial evaluation: Exploiting rule structure in probabilistic inference. In *IJCAI-97*, pages 1284–1291, Nagoya, Japan, 1997a. URL http://www.cs.ubc.ca/spider/poole/abstracts/pro-pa.html.

D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94:7–56, 1997b. URL http://www.cs.ubc.ca/spider/poole/abstracts/icl.html. special issue on economic principles of multi-agent systems.

D. Poole. Abducing through negation as failure: stable models in the Independent Choice Logic. *Journal of Logic Programming*, 44(1–3):5–35, 2000b. URL http://www.cs.ubc.ca/spider/poole/abstracts/abnaf.html.

D. Poole and N. L. Zhang. Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence Research*, 18:263–313, 2003.

D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, New York, 1998.

J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Computing*, 13(3,4):287–312, 1995.

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

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

L. J. Savage. *The Foundation of Statistics*. Dover, New York, 2nd edition, 1972.

M. Shanahan. Prediction is deduction, but explanation is abduction. In *IJCAI-89*, pages 1055–1060, Detroit, MI, Aug. 1989.

S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, 2000. IEEE.

N. L. Zhang and D. Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, Dec. 1996.