Introduction to BLOG Syntax

Brian Milch

August 8, 2007

This document is an excerpt from Chapter 4 of my Ph.D. dissertation. It begins with some examples of BLOG models, and then gives a thorough description of BLOG syntax.

1 Examples

Our first BLOG model is for the urn-and-balls scenario introduced in Chapter 3. In this scenario, we have an urn containing a finite but unknown number of balls, each of which is either blue or green. We repeatedly draw a ball from the urn, observe its color (with some probability of error), and return it to the urn. Given these observations, we may want to obtain a posterior probability distribution on the number of balls in the urn, or find the posterior probability that the first and second draws yielded the same ball.

Figure 1 shows a BLOG model for this scenario. The first 6 lines of this model define the particular logical language whose model structures are to serve as possible worlds. Line 1 says that the language contains three types: Color, Ball, and Draw. Lines 2–4 say that the language includes three function symbols, TrueColor, BallDrawn and ObsColor, which are all random: their interpretations vary from world to world. The argument types and return types of these functions are given in a syntax reminiscent of C or Java. Line 5 asserts that two colors, denoted by the constant symbols Blue and Green, are guaranteed to exist in all possible worlds; line 6 makes a similar assertion about draws.

In the hypothetical stochastic process for constructing a possible world, we begin with just the colors and draws that are guaranteed to exist. Then the *number statement* on line 7 says that we add a random number of balls, sampled according to a Poisson distribution with mean 6. Line 8 says that for each ball b, TrueColor(b) is sampled according to a probability table that puts probability 0.5 on Blue and 0.5 on Green (the ordering of the probabilities in the vector $[0.5, 0.5]$ corresponds to the order in which Blue and Green were introduced on line 5). Next, for each draw d,

```
1   type Color; type Ball; type Draw;

2   random Color TrueColor(Ball);
3   random Ball BallDrawn(Draw);
4   random Color ObsColor(Draw);

5   guaranteed Color Blue, Green;
6   guaranteed Draw Draw1, Draw2, Draw3, Draw4;

7   #Ball ~ Poisson[6]();

8   TrueColor(b) ~ TabularCPD[[0.5, 0.5]]();

9   BallDrawn(d) ~ Uniform({Ball b});

10  ObsColor(d)
11      if (BallDrawn(d) != null) then
12          ~ TabularCPD[[0.8, 0.2], [0.2, 0.8]]
13              (TrueColor(BallDrawn(d)));
```

Figure 1: BLOG model for balls in an urn, with four draws.

line 9 tells us to sample the value of $\mathsf{BallDrawn}(d)$ uniformly from the set of balls that exist. Finally, lines 10–13 tell us how to sample the observed color for each ball d. If $\mathsf{BallDrawn}(d) \neq \mathsf{null}$, then $\mathsf{ObsColor}(d)$ is sampled according to a tabular CPD that conditions on $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$. The case where $\mathsf{BallDrawn}(d) = \mathsf{null}$ occurs only when the number of balls in the world happens to be zero; in this case, $\mathsf{ObsColor}(d)$ gets a default value of null. The statements on lines 8 and 10–13 are called *dependency statements*.

This generative process defines a prior probability distribution over possible worlds. If we condition on observed values for certain random variables, such as $(\mathsf{ObsColor}(\mathsf{Draw1}) = \mathsf{Blue}, \mathsf{ObsColor}(\mathsf{Draw2}) = \mathsf{Blue})$, then a query event such as $\mathsf{BallDrawn}(\mathsf{Draw1}) = \mathsf{BallDrawn}(\mathsf{Draw2})$ has a well-defined posterior probability. Computing posterior probabilities is the task of the inference algorithms that we discuss in Chapter 5.

Our next BLOG model, shown in Figure 2, is for the hurricane scenario first discussed in Chapter 3. The premise for this scenario is that a hurricane is going to strike two cities, A and B, but it is unknown which city will be struck first. The level of damage that each city sustains depends on its level of preparations; also, the

```
1   type City; type PrepLevel; type DamageLevel;

2   random City First;
3   random PrepLevel Prep(City);
4   random DamageLevel Damage(City);

5   guaranteed City A, B;
6   guaranteed PrepLevel High, Low;
7   guaranteed DamageLevel Severe, Mild;

8   First ~ Uniform({City c});

9   Prep(c)
10      if First = c then ~ TabularCPD[[0.5, 0.5]]
11      else ~ TabularCPD[[0.9, 0.1], [0.1, 0.9]](Damage(First));

12  Damage(c) ~ TabularCPD[[0.2, 0.8], [0.8, 0.2]](Prep(c));
```

Figure 2: A BLOG model for the hurricane scenario.

level of preparations in the second city to be hit depends on the level of damage in the first. The logical language used for this scenario has three types, City, PrepLevel and DamageLevel, and three random functions, First, Prep and Damage. Note that First is a random zero-ary function; in other words, it is a constant symbol whose denotation is random. In this model, the only objects that exist are guaranteed objects: the cities A and B, the preparation levels High and Low, and the damage levels Severe and Mild.

Line 8 says that the denotation of First is chosen uniformly at random from the set of cities. Lines 9–11 give the probability model for Prep(c): if c is the first city hit, then Prep(c) has a uniform 0.5–0.5 prior distribution; otherwise, Prep(c) depends on Damage(First) through a tabular CPD. Finally, line 12 says that Damage(c) depends on Prep(c) through a certain CPD.

Our next example is a simplified version of the bibliographic citation matching problem [Lawrence *et al.*, 1999; Pasula *et al.*, 2003].

Example 1. *Suppose we have collected a set of citation strings from the works cited sections of online papers. The citations may be in many different formats; they may use various abbreviations; and they may contain typographical errors. The task is to construct a database containing exactly one record for each publication that is cited and each researcher who is mentioned in these citations. This database should specify*

the correct names and titles of the entities, as well as the mapping from publications to their authors.

A possible world for this scenario includes both the citations, which we observe, and the researchers and publications that underlie them. We can think of these worlds as model structures of a first-order language with types Researcher, Publication, Citation and String (the String type is actually built into BLOG). This language includes a function Name that maps researchers to strings; a function Title that maps publications to strings, and a function Author that maps publications to researchers. We also have a function PubCited that maps citations to the publications they refer to, and a function Text that maps citations to their observed text strings.

Figure 3 shows a BLOG model for this example, based on the model in [Pasula *et al.*, 2003]. The model makes the simplifying assumption that in the academic field we are dealing with, there is a pool of researchers who are all equally likely to write papers. Line 9 describes the step that generates these researchers; the number of researchers is chosen from a CPD called `NumResearchersPrior`. Similarly, we assume that there is a pool of publications (generated by line 10) that are all equally likely to be cited. The name of each researcher and the title of each publication are sampled from certain prior distributions (lines 11 and 12). Each publication has some number of authors; each author is sampled uniformly from the pool of researchers (lines 14–15). Then, for each citation, the publication cited is chosen uniformly from the pool of publications (line 16). The text of a citation c depends on the title of PubCited(c) and the names of its authors.

Note that under this model, a possible world may contain publications that are not referred to by any citation (and also researchers that are not authors of any publication). This is realistic: a finite collection of citations may not cover all the publications that could be cited. This model also ensures that questions such as, "What is the probability that an author named "Leslie Kaelbling" has written a paper that is not cited in our collection?" have well-defined answers. One might worry that if our queries are just about cited publications, then having a potentially large number of uncited publications in our possible worlds would make inference slower than necessary. However, as we will see in Chapter 5, there are inference algorithms that simply ignore the attributes of such irrelevant objects.

For our final example, we move from citations to radar blips. We describe a fairly standard version of the multitarget tracking problem [Bar-Shalom and Fortmann, 1988].

Example 2. *An unknown number of aircraft exist in some volume of airspace. An aircraft's state (position and velocity) at each time step depends on its state at the previous time step. We observe the area with radar: aircraft may appear as identical*

```
1   type Researcher; type Publication; type Citation;

2   random String Name(Researcher);
3   random String Title(Publication);
4   random NaturalNum NumAuthors(Publication);
5   random Researcher NthAuthor(Publication, NaturalNum);
6   random Publication PubCited(Citation);
7   random String Text(Citation);

8   guaranteed Citation Cit1, Cit2, Cit3, Cit4;

9   #Researcher ~ NumResearchersPrior();
10  #Publication ~ NumPubsPrior();

11  Name(r) ~ NamePrior();

12  Title(p) ~ TitlePrior();
13  NumAuthors(p) ~ NumAuthorsPrior();
14  NthAuthor(p, n)
15      if n < NumAuthors(p) then ~ Uniform({Researcher r});

16  PubCited(c) ~ Uniform({Publication p});
17  Text(c) ~ NoisyCitationGrammar
18              (Title(PubCited(c)),
19               {n, Name(NthAuthor(PubCited(c), n))
20                for NaturalNum n :  n < NumAuthors(PubCited(c))});
```

Figure 3: BLOG model for Example 1 with four observed citations.

blips on a radar screen. Each blip gives the approximate position of the aircraft that generated it. However, some blips may be false detections, and we may not get a blip for every aircraft at every time step. The questions we would like to answer include: What aircraft exist? What are their trajectories? Are there any aircraft that have not been observed?

The possible worlds of this scenario specify the trajectories of all the aircraft over time, as well as the time stamps and locations of all radar blips that appear. We will model the states of the aircraft at an infinite sequence of time steps $0, 1, 2, \ldots$, although events at "future" time steps after our last observation will be irrelevant for inference (unless we explicitly ask queries about the future). These possible worlds

can be represented as model structures of a first-order language with types Aircraft and Blip, as well as built-in types that represent natural numbers (time steps) and real vectors. The trajectories of the aircraft are represented by a function State(a, t) which maps aircraft a and natural numbers t to six-dimensional real vectors (three dimensions for the aircraft's position and three for velocity). A function MeasuredPos from blips to $\mathbb{R}^3$ represents the range, azimuth, and elevation that are measured for each blip. There is also a function Source that maps each blip to the aircraft that generated it; false detection blips have Source values of null. Finally, a function Time maps each blip to the time when it appeared (a natural number).

```
1   type Aircraft; type Blip;

2   random R6Vector State(Aircraft, NaturalNum);
3   random R3Vector MeasuredPos(Blip);

4   origin Aircraft Source(Blip);
5   origin NaturalNum Time(Blip);

6   #Aircraft ~ NumAircraftPrior();

7   State(a, t)
8       if t = 0 then ~ InitState()
9       else ~ StateTransition(State(a, Pred(t)));

10  #Blip(Source = a, Time = t) ~ DetectionCPD(State(a, t));
11  #Blip(Time = t) ~ NumFalseAlarmsPrior();

12  MeasuredPos(b)
13      if (Source(b) = null) then ~ FalseAlarmDistrib()
14      else ~ MeasurementCPD(State(Source(b), Time(b)));
```

Figure 4: BLOG model for Example 2.

The BLOG model for this scenario (Figure 4) describes the following process: first, sample the number of aircraft in the area. Then, for each time step t (starting at $t = 0$), choose the state of each aircraft given its state at time $t - 1$. Also, for each aircraft a and time step t, possibly generate a radar blip b with Source$(b) = a$ and Time$(b) = t$. Whether a blip is generated or not depends on the state of the aircraft. Also, at each step t, generate some false-alarm blips b' with Time$(b') = t$ and Source$(b') =$ null. Note that the *origin functions* Source and Time are set on

each blip when it is generated; they are not set in separate random sampling steps. Finally, sample the position for each blip given the true state of its source aircraft (or using a default distribution for a false-alarm blip).

This generative process is different from those in the previous examples in that the objects of each type are not all created in a single step. Here, there are many generative steps that add blips to the world: one for each aircraft a and each time t, plus additional steps that generate false-alarm blips. Intuitively, objects are generating other objects. Furthermore, the numbers of blips generated can depend on the values of the State function, so the probability model does not easily decompose into a portion that defines a distribution for what objects exist and a portion that defines distributions for attribute values given object existence. Uncertainty about what objects exist is fully integrated with uncertainty about attributes and relations.

2 Syntax

This section provides a more formal definition of BLOG syntax. A BLOG model consists of a sequence of statements, separated by semicolons. Thus, for example, line 1 of Figure 1 contains three statements, while lines 10–13 of that figure constitute a single statement. There are six kinds of statements in BLOG: type declarations, function declarations, guaranteed object statements, nonrandom function definitions, dependency statements, and number statements. A BLOG model M defines a typed first order language $\mathcal{L}_M$, a set of possible worlds Ω_M, and (if it is well-defined) a probability measure P_M on Ω_M.

2.1 Built-in types and functions

The language defined by a BLOG model always includes a set of built-in types, summarized in Table 1. Each built-in type has a fixed extension in all possible worlds: for instance, the extension of NaturalNum is always the natural numbers. For most built-in types, there are also built-in constant symbols that denote objects of that type. Numerals without decimal points are constant symbols denoting objects of type NaturalNum; numerals with decimal points denote objects of type Real; strings enclosed in double quotes denote objects of type String. The interpretations of these constant symbols are fixed across all possible worlds.

Types representing real-valued vectors are a special case. For every natural number $k \geq 2$, there is a built-in type RkVector; examples include R2Vector, R3Vector, etc. Vectors are denoted not by constant symbols, but by terms using built-in functions, which we will introduce below.

The Real type and the RkVector types have uncountable extensions, but our

Type symbol	Extension	Examples of built-in constants
Boolean	{true, false}	`true`, `false`
NaturalNum	$\mathbb{N}$	`0`, `1`, `2`
Real	$\mathbb{R}$	`3.14`, `-2.7`, `1.0`
RkVector (for $k \geq 2$)	$\mathbb{R}^k$	
String	Finite character strings	`""`, `"hello"`, `"R2-D2"`

Table 1: Built-in types, their extensions, and some illustrative built-in constant symbols denoting objects of each type. The Real type and the RkVector types are not included in discrete BLOG.

Function symbol	Arguments	Return type	Shorthand
Succ	NaturalNum n	NaturalNum	
Pred	NaturalNum n	NaturalNum	
Sum	NaturalNum n, NaturalNum m	NaturalNum	n + m
Diff	NaturalNum n, NaturalNum m	NaturalNum	n - m
LessThan	NaturalNum n, NaturalNum m	Boolean	n < m
GreaterThan	NaturalNum n, NaturalNum m	Boolean	n > m
LessThanOrEqual	NaturalNum n, NaturalNum m	Boolean	n <= m
GreaterThanOrEqual	NaturalNum n, NaturalNum m	Boolean	n >= m
ConstructRkVector	Real r_1, ..., Real r_k	RkVector	[r_1, ..., r_k]
Concat	String s_1, String s_2	String	

Table 2: Built-in functions, their argument types (with variables standing for the arguments) and return types, and any special syntax that is used for them.

development of PBMs in Chapter 3 is limited to discrete variables. This discrepancy reflects a difference between our implemented BLOG engine, which supports models with continuous variables, and our theoretical results, which do not yet apply to such models. We discuss continuous types here to show that BLOG syntax is general enough to handle them. However, our formal results in this thesis apply only to *discrete BLOG*, a version of BLOG without the Real and RkVector types.

We now move on to BLOG's built-in functions, listed in Table 2. Like the built-in types, these functions are included in the language defined by each BLOG model. The built-in functions include Succ and Pred, which yield the successor and predecessor of a natural number (Pred(0) returns null). There are also built-in functions Sum and Diff on natural numbers; Diff(n, m) evaluates to null when n is less than m. Standard comparison functions such as LessThan are also built in. Furthermore, rather than requiring modelers to write terms such as `LessThan(3, 4)`, BLOG supports infix operators as a shorthand. Thus, one can use terms such as `3 < 4`. Next,

for each natural number $k \geq 2$, BLOG includes a function ConstructRkVector that takes real numbers $r_1, \ldots, r_k$ and returns the vector $(r_1, \ldots, r_k)$. The shorthand way to invoke such a function is to include k real-valued terms in square brackets: for example, `[0.2, 1.9, -2.0]`. The advantage of treating these expressions as function applications rather than just large constant symbols is that the vector elements can be represented by arbitrary terms, not just numerals. Finally, there is a built-in function Concat that returns the concatenation of two strings.

BLOG currently has no built-in constructs for referring to lists of objects, except for vectors of real numbers. As we saw in the citation model (Figure 3, we can represent lists using functions that take a natural number as an argument, such as NthAuthor(p, n). However, lists are not treated as objects, and thus cannot serve as arguments or values of functions. This limitation may be remedied in a future version of BLOG.

2.2 Type and function declarations

Type and function declarations do most of the work of defining a domain-specific logical language for a BLOG model (guaranteed object statements do some of this work as well). A *type declaration* has the form:

`type` τ`;`

where τ is an identifier (a string of alphanumeric characters) that will serve as the symbol for the type. A *function declaration* has one of the following forms:

`random` r f`(`$a_1, \ldots, a_k$`);`
`nonrandom` r f`(`$a_1, \ldots, a_k$`);`
`origin` r f`(`a`);`

Here f is an identifier that will serve as symbol for the function, r is a previously declared type symbol identifying the function's return type, and $a_1, \ldots, a_k$ are previously declared type symbols identifying the function's argument types. If $k = 0$ — that is, if f is a constant symbol — then the parentheses may be omitted. The keywords `random`, `nonrandom` and `origin` specify how a function's values are determined. *Random functions* are those whose values vary across possible worlds; they have their values set (on each tuple of arguments) by steps in the generative process. The values of *nonrandom functions* are fixed across all worlds. *Origin functions* play a special role in scenarios where objects generate other objects, as discussed in Sec. 2.6 below. Note that an origin function must take exactly one argument.

2.3 Guaranteed object statements

A *guaranteed object statement* asserts that certain objects are known to exist and to be distinct, and also declares constant symbols for these objects. For instance, line 5 of Figure 1 asserts that in every possible world, there are two distinct objects of type Color denoted by the constant symbols Blue and Green. The general form of a guaranteed object statement is:

`guaranteed` τ $c_1, \ldots, c_n$`;`

where τ is a user-defined (*i.e.*, not built-in) type symbol and $c_1, \ldots, c_n$ are identifiers that will serve as constant symbols. A BLOG model may contain multiple guaranteed object statements for the same type: all the constant symbols in all these statements denote distinct objects of the given type (it is an error to include the same constant symbol more than once). Thus, a single guaranteed object statement is not necessarily an exhaustive list of the guaranteed objects of a given type. However, the full set of guaranteed object statements in a model is exhaustive, in that each object that exists in any possible world must be introduced in some guaranteed object statement or be generated by some number statement. The order in which guaranteed objects are introduced is relevant: CPDs such as `TabularCPD` interpret their probability parameters as corresponding to guaranteed objects in they order they were declared.

BLOG also includes a shorthand syntax for defining many guaranteed objects at once. One can write a statement of the form:

`guaranteed` τ c`[`n`];`

to define n guaranteed objects of type τ, denoted by the symbols $c1, c2, c3, \ldots$. For instance, we could abbreviate line 6 in Figure 1 as:

```
guaranteed Draw Draw[4];
```

The guaranteed object statements for type τ in M jointly define a set of guaranteed objects $\mathrm{Guar}_M(\tau)$. We let the constant symbols in these statements serve as their own denotations, so $\mathrm{Guar}_M(\tau)$ is just the set of constant symbols introduced by guaranteed object statements for type τ.[1] For built-in types τ, $\mathrm{Guar}_M(\tau)$ is the same in every model M: it is the extension given in Table 1. The *object set* for type τ in model M is:

$$\mathcal{O}_M(\tau) \triangleq \bigcup_{\omega \in \Omega_M} [\tau]^\omega$$

[1] This does not preclude the possibility that some other constant symbols — introduced in function declarations rather than guaranteed object statements — might end up denoting the same objects.

Because the set of objects of type τ can vary from world to world, $\mathcal{O}_M(\tau)$ may be a strict superset of $\text{Guar}_M(\tau)$. The *non-guaranteed objects* of type τ in model M are:

$$\text{NonGuar}_M(\tau) \triangleq \mathcal{O}_M(\tau) \setminus \text{Guar}_M(\tau)$$

We will also write NonGuar_M for the union of $\text{NonGuar}_M(\tau)$ over all types τ in $\mathcal{L}_M$.

2.4 Nonrandom function definitions

We mentioned above that a function symbol can be declared as `nonrandom`, meaning that it has the same interpretation in all possible worlds. Nonrandom function symbols serve two main purposes. First, they may represent mathematical functions that are not built into BLOG. For instance, in the aircraft tracking example, we could include a nonrandom function IsInSphere(s, r), which takes an aircraft state vector s and returns true if the position portion of that vector is in the sphere of radius r around the origin. However, nonrandom function symbols can also be used to represent known, scenario-specific information about guaranteed objects. For instance, suppose that in the urn-and-balls scenario, each draw from the urn is performed by some person; perhaps the identity of that person influences the error rate for ObsColor. If we know which person performed each draw, then there is no reason to define a prior probability model for the function Agent(d) that returns the person responsible for draw d. Instead, we can let Agent be nonrandom.

Considering the broad range of nonrandom functions that one might want to define, we do not attempt to include general syntax for specifying interpretations in BLOG. Existing programming languages are fine tools for specifying nonrandom functions. Since our BLOG engine is implemented in Java, we specify functions using instances of Java classes that implement an interface called `FunctionInterp`. A *nonrandom function definition* binds a nonrandom function symbol to a Java `FunctionInterp` object, using the syntax:

`nonrandom` f `= interp` c`[`p_1`, ...,` p_n`];`

Here f is a k-ary function that has already been declared, c is the name of a Java class that implements `FunctionInterp`, and $p_1, \ldots, p_n$ are expressions that serve as parameters to c. For example, to specify the interpretation for Agent, we could write:

```
nonrandom Agent = interp DrawToPersonInterp["agents.dat"];
```

Here we are assuming that `agents.dat` is a file containing the mapping from draws to people in some format, and `DrawToPersonInterp` is a class that can read that format and define a function from guaranteed objects of type Draw to guaranteed objects of type Person. When the BLOG engine processes this statement while

loading a model, it creates an instance of class `DrawToPersonInterp`, passing the string `"agents.dat"` to that instance's constructor.

The expressions $p_1, \ldots, p_n$ that serve as parameters may be terms or formulas of $\mathcal{L}_M$ that are well-formed in the empty scope, or set expressions (see Section 2.8). These expressions must be *syntactically nonrandom*: that is, they must not contain any random function symbols, and must not include quantifiers or set expressions ranging over types that have number statements in M. Furthermore, it must be possible to compute the values of all the parameters without invoking the interpretation of the function f — that is, the definitions of nonrandom functions must be acyclic (the BLOG engine checks for cyclic definitions and gives error messages when they occur). If there are no parameters in a nonrandom function definition, then the square brackets after the class name can be omitted.

BLOG includes a standard `FunctionInterp` class called `ConstantInterp` that can serve as an interpretation for zero-ary function symbols (that is, constant symbols). The `ConstantInterp` class expects one parameter: a term specifying the value of the constant symbol. In fact, BLOG includes a special syntax for definitions that use `ConstantInterp`. A statement of the form:

`nonrandom` f `=` t`;`

where f is a zero-ary function and t is a syntactically nonrandom term, is an abbreviation for:

`nonrandom` f `= interp ConstantInterp[`t`];`

BLOG also allows nonrandom functions to be declared and defined in a single statement. For instance, the statement:

```
nonrandom Person Agent(Draw d)
       = interp DrawToPersonInterp["agents.dat"];
```

is both a declaration and a definition of the function Agent. A BLOG model must contain exactly one function definition statement for each nonrandom function that it declares.

We can also give a more mathematical definition of a nonrandom function interpretation that does not depend on any particular implementation language (such as Java).

Definition 1. *In a BLOG model M, a* nonrandom function interpretation *for a function with type signature $(r, a_1, \ldots, a_k)$ is a function from* $\mathrm{Guar}_M(a_1) \times cdots \times \mathrm{Guar}_M(a_k)$ *to* $\mathrm{Guar}_M(r) \cup \{\mathsf{null}\}$.

We will use $[f]_M$ to denote the interpretation that M assigns to a nonrandom function f.

2.5 Dependency statements

Dependency statements specify probability distributions for the values of random functions. As an example, consider the dependency statement for $\mathsf{State}(a, t)$ in Figure 4:

```
State(a, t)
    if t = 0 then ~ InitState()
    else ~ StateTransition(State(a, Pred(t)));
```

In the generative process, this statement is applied for every Aircraft object a and every natural number t. If $t = 0$, the conditional distribution for $\mathsf{State}(a, t)$ is given by the *elementary CPD* `InitState`; otherwise it is given by the elementary CPD `StateTransition`, which takes $\mathsf{State}(\mathsf{a}, \mathsf{Pred}(\mathsf{t}))$ as an argument. These elementary CPDs define distributions over objects of type R6Vector (the return type of State). In our implementation, elementary CPDs are instances of Java classes that implement an interface called `CondProbDistrib`.

A BLOG model contains exactly one dependency statement for each random function (the order of these statements is irrelevant). The general syntax for a dependency statement is as follows:

f(`(`x_1`, ..., `x_k`)`)
    `if` φ_1 `then ~` c_1`(`e_{11}`, ..., `e_{1n_1}`)`
    `elseif` φ_2 `then ~` c_2`(`e_{21}`, ..., `e_{2n_2}`)`
    ⋮
    `else ~` c_m`(`e_{m1}`, ..., `e_{mn_m}`);`

The statement begins with a header, which includes the random function symbol f and logical variables $x_1, \ldots, x_k$ that stand for f's arguments. This function f is called the *child function* in the dependency statement (by analogy to the child variable for a CPD in a Bayesian network). The header defines a scope $\beta_f = \{(x_1, a_1), \ldots, (x_k, a_k)\}$, where $a_1, \ldots, a_k$ are the types of f's arguments.

The remainder of the statement defines a sequence of *clauses*. A clause consists of a condition φ, an elementary CPD c, and a tuple of expressions $(e_1, \ldots, e_n)$ that serve as arguments to the CPD. The syntactic representation of a clause begins with `if` for the first clause, `elseif` for subsequent clauses, and `else` for the last clause: as these keywords suggest, the clause that is active in a particular world is the first clause whose condition is satisfied. The condition in a clause can be any formula of $\mathcal{L}_M$ that is well-formed in the scope β_f. The condition for the last clause is not specified explicitly: it is always simply true. If desired, the set of clauses (that is, the portion of the dependency statement between the header and the final semicolon) can be enclosed in curly braces for clarity.

After the condition, each clause specifies an elementary CPD for the range $\mathcal{O}_M(\text{ret}f) \cup \{\mathsf{null}\}$. This specification can consist of just a Java class name, such as `StateTransition`, or it may consist of a Java class name followed by some parameters in square brackets. These parameters are not to be confused with the CPD arguments that come after the elementary CPD; for instance, in:

```
TabularCPD[[0.8, 0.2], [0.2, 0.8]](TrueColor(BallDrawn(d)))
```

the vectors $[0.8, 0.2]$ and $[0.2, 0.8]$ are parameters, whereas $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$ is an argument. The parameters must be nonrandom, and cannot depend on the function arguments $x_1, \ldots, x_k$. Formally, the parameters may be any syntactically nonrandom terms, formulas, or set expressions that are well-formed in the empty scope. When a dependency statement is processed by the BLOG engine, the parameters of each elementary CPD are evaluated, and their values are passed to the constructor of the given Java class to create a new instance. For example, if a model includes two elementary CPD specifications `Poisson[6]` and `Poisson[8]`, then the engine creates two instances of the Java class `Poisson`, one with mean 6 and one with mean 8. These Java objects define elementary CPDs, in a sense that we make precise in Section 2.9.

Finally, a clause specifies a tuple of CPD arguments $e_1, \ldots, e_n$. Unlike CPD parameters, which are evaluated at initialization time without reference to any particular world, CPD arguments take on values that depend on the possible world and the assignment of values to the variables in β_f. The arguments can be any terms, formulas, or set expressions that are well-defined in β_f. Examples of arguments include $\{\mathsf{Ball}\ b\}$ and $\mathsf{Name}(\mathsf{Author}(\mathsf{PubCited}(c)))$ in Figure 1, and $\mathsf{State}(a, t)$ in Figure 4.

We have given the syntax for fully general dependency statements, but our example models rarely use this full-fledged if-then-else form: they use certain abbreviations. The allowed abbreviations are as follows.

- If we want to define a dependency model with just a single clause whose condition is `true`, we can dispense with the `if` and `then` keywords and write a statement of the form:

  f`(`x_1`, ..., `x_k`)` $\sim$ c`(`e_1`, ..., `e_n`);`

  Line 12 of Figure 2 is a good example of this format.

- There is a special `CondProbDistrib` class called `EqualsCPD` that is used to represent deterministic dependencies. `EqualsCPD` takes a single expression e as an argument, and constrains the child function to take the value of that

expression with probability one. The expression e may be a term, in which case the return type of the child function f must be the same as the type of the term; a formula, in which case ret(f) must be Boolean; or a cardinality expression (see Section 2.8 below), in which case ret(f) must be NaturalNum. Instead of writing $\sim$ `EqualsCPD(`e`)`, we can write `=` e as a shorthand. For example, in a version of the urn-and-balls scenario with deterministic observations, we could write:

```
ObsColor(d) = TrueColor(BallDrawn(d));
```

- If a dependency statement contains one or more clauses with the `if` and `elseif` keywords but no `else` clause, then there is an implicit final clause of the form:

  `else =` *default*

  where *default* is false when the child function is Boolean, and null when the child function has any other return type.

- If the child function has no arguments, then we can omit the empty parentheses. We see this in the dependency statement for the random constant symbol First in Figure 2. Similarly, if an elementary CPD takes no arguments, the empty parentheses can be omitted after it.

Finally, just as we can combine function declarations with nonrandom function definitions, BLOG also allows us to combine function declarations with dependency statements. We simply add the `random` keyword and type information to the header. So, for example, we could combine lines 3 and 9 in Figure 1 to yield:

```
random Ball BallDrawn(Draw d) ~ Uniform({Ball b});
```

2.6 Number statements

Number statements specify how objects are added to the world in the generative process described by a BLOG model. In the simple case, a model contains at most one number statement for each type: then each number statement defines a distribution for the total number of non-guaranteed objects of that type. However, as we saw in the aircraft tracking example (Figure 4), it is not always convenient to assume that all the non-guaranteed objects of a given type are added to the world in a single generative step. The radar blips in this example are generated in many separate steps: one for each aircraft at each time point, plus another step at each time point to generate "false alarm" blips. We can think of blips as being generated

by other objects — namely aircraft and natural numbers (time points) — and being tied back to those generating objects by the origin functions Source and Time. Line 10 in Figure 4 also specifies that the number of blips generated by aircraft a at time t depends on $\mathsf{State}(a, t)$.

Thus, the general syntax of a number statement is as follows:

```
#τ(g_1 = x_1, ..., g_k = x_k)
    if φ_1 then ~ c_1(e_11, ..., e_1n_1)
    elseif φ_2 then ~ e_2(e_21, ..., e_2n_2)
    ⋮
    else ~ c_m(e_m1, ..., e_mn_m);
```

This is the same syntax as for dependency statements, except for the header. Here, the header consists of a user-defined type symbol τ, a sequence of distinct origin function symbols $g_1, \ldots g_k$ that take an argument of type τ, and a sequence of logical variables $x_1, \ldots, x_k$ that stand for the generating objects, *i.e.*, the return values of the origin functions. The header defines a scope $\{(x_1, r_1), \ldots, (x_k, r_k)\}$, where $r_1, \ldots, r_k$ are the origin functions' return types; expressions in the rest of the number statement are evaluated in this scope. As usual, if a number statement involves no origin functions, the empty parentheses can be omitted.

The header for a number statement does not need to include all the origin functions for a given type: for example, line 11 in Figure 4 does not include the function Source. When an origin function is not included, its value on the generated objects defaults to null. A BLOG model can contain any number of number statements for a given type, as long as no two number statements use the same set of origin functions. This restriction ensures that one can always tell which number statement generated a non-guaranteed object o by looking at the values of the origin functions on o.

After the header, a number statement defines a sequence of clauses, just as in a dependency statement. The abbreviations that apply to dependency statements (omitting "`if`...`then`" when the condition is just true, the implicit `else` clause, etc.) can also be used in number statements. The only differences are that all the elementary CPDs used in a number statement must have $\mathbb{N}$ as their range set, and the default value specified by the implicit `else` clause is zero rather than false or null.

Note that the number of objects added in any single generative step is always finite. However, object generation can be recursive: objects can generate other objects of the same type. For instance, consider a model of sexual reproduction in which every male–female pair of individuals produces some number of offspring. We could represent this with the number statement:

```
#Individual(Mother = m, Father = f)
        if Female(m) & !Female(f) then ~ NumOffspringPrior;
```

If a model contains recursive number statements, the total number of non-guaranteed objects in a possible world may be infinite, even though each generative step adds a finite number of them.

Another way to obtain an infinite set of non-guaranteed objects is by letting them be generated by the natural numbers. For instance, to model a version of the urn-and-balls scenario with infinitely many draws, we could use the following statements instead of the guaranteed object statement for draws:

```
origin NaturalNum Index(Draw); #Draw(Index = n) = 1;
```

These statements ensure that for each natural number n, there is exactly one draw whose index is n (recall that "`= 1`" is an abbreviation for "$\sim$ `EqualsCPD(1)`"). However, the statements do not give us any constant symbols, or even any terms, that we can use to refer to draws. We can remedy this with the following statements:

```
random Draw NthDraw(NaturalNum);
NthDraw(n) ~ Iota({Draw d:  Index(d) = n});
```

Here `Iota` is an elementary CPD that takes in a set of size one, and defines a distribution that puts probability one on the sole element of that set (if the given set is not of size one, it puts probability one on null). Once these statements are included, we can refer to draws using terms such as NthDraw(14). This is admittedly a rather complicated solution for a simple modeling task; future versions of BLOG may include syntax to make this more straightforward.

2.7 Logical formulas

We have said in a BLOG model M, the formulas of the logical language $\mathcal{L}_M$ can serve as constituents in nonrandom function definitions, dependency statements, and number statements. However, when we defined first-order logical languages in Chapter 2, we used non-ASCII characters such as $\neg$, $\wedge$, and $\forall$. Table 3 shows how we replace such characters with ASCII equivalents.

2.8 Set expressions

In this section, we give an overview of the set expressions that BLOG supports, followed by a formal definition of their syntax and semantics. As we shall see, the term "set expression" is something of a misnomer: while some of these expressions do denote sets, others denote multisets, and one kind of set expression denotes a number that is the cardinality of a set. However, "set expression" remains a convenient way of referring to these various expressions that use set notation.

Logical syntax	BLOG syntax
$t_1 = t_2$	t_1 `=` t_2
$t_1 \neq t_2$	t_1 `!=` t_2
$\neg\psi$	`!`ψ
$\psi \wedge \chi$	ψ `&` χ
$\psi \vee \chi$	ψ `|` χ
$\psi \rightarrow \chi$	ψ `->` χ
$\forall\ \tau\ v\ \psi$	`forall` τ v ψ
$\exists\ \tau\ v\ \psi$	`exists` τ v ψ

Table 3: Standard syntax and BLOG syntax for logical formulas.

We have seen several places in our examples where a set expression is used as an argument to an elementary CPD. The first of these is on line 9 of Figure 1, where we pass {`Ball b`} into the `Uniform` CPD. In an alternative version of the urn-and-balls model where there are multiple urns, we could use a set expression such as {`Ball b : In(b, Urn1)`}. In these cases, the goal is to define a distribution over a set of objects that varies from world to world; the CPD needs to take this set as an argument so that it can define the desired distribution.

The `Uniform` CPD selects uniformly from a given set, but what if we want a CPD to do weighted sampling? For example, we might want to model a scenario where the chance that a ball is selected on any given draw depends on its color. In this case, we could use an elementary CPD `SampleGivenColor` that takes not just a set of balls, but a set of pairs (b, c) where b is a ball and c is its color. The BLOG syntax for this is:

```
{b, TrueColor(b) for Ball b}
```

We call this kind of expression a *tuple multiset expression*; it defines a multiset of tuples. A *multiset* U consists of a set set_U and a function $\text{mult}_U : \text{set}_U \rightarrow \mathbb{N}$ that returns a *multiplicity* for each element of set_U, representing the number of times that element occurs. In our example, the multiplicity of each tuple is one, because each value of the logical variable b yields a different tuple. But if the expression were simply {`TrueColor(b) for Ball b`}, several values of b could yield the same tuple (here each tuple has just one element, which is a color). If U is a multiset, we will simply write $o \in U$ to mean $o \in \text{set}_U$. Also, if S is a set, we will use the notation $\{|f(x) : x \in S|\}$ to represent the multiset U with $\text{set}_U = \{f(x) : x \in S\}$ and $\text{mult}_U(o) = |\{x \in S : f(x) = o\}|$. We will use the same notation to define one multiset in terms of another one: if U is a multiset, then $\{|f(x) : x \in U|\}$ is the multiset U' with $\text{set}_{U'} = \{f(x) : x \in U\}$ and $\text{mult}_{U'}(o) = \sum_{(x \in U\,:\, f(x) = o)} \text{mult}_U(x)$.

Set expressions are also useful when a conditional distribution depends on the attributes of a set of objects. For instance, lines 17–20 in Figure 3 say that the text

of a citation c depends on the names of all the authors of PubCited(c). The tuple multiset expression used here is:

```
{n, Name(NthAuthor(PubCited(c), n))
                    for NaturalNum n : n < NumAuthors(PubCited(c))}
```

This expression evaluates to a multiset of pairs (n, s) where n is a natural number and s is a string (an author name); these pairs help determine the distribution over citation strings. The numbers need to be included in these pairs so that the distribution over citation strings reflects the order of the authors (a future version of BLOG may include a new kind of set expression that evaluates to a list rather than a multiset). CPDs with tuple multisets as arguments can also perform more standard aggregation operations, such as taking an average or median of a set of real numbers. For instance, suppose the chance of a paper being accepted to a conference depends on the average intelligence of its authors. If Intelligence is a function from Researcher to Real, then we can write the dependency statement:

```
Accepted(p) ~ AcceptanceCPD({Intelligence(NthAuthor(p, n))
                                for NaturalNum n : n < NumAuthors(p)});
```

This `AcceptanceCPD` could, for example, take the average of the real numbers passed into it, and pass this average through a logistic function to get the probability of acceptance. It might be useful in future version of BLOG to allow aggregation functions (such as Average) to be separated from CPDs, so elementary CPDs and aggregation functions could be mixed and matched in a compositional way. Currently, however, BLOG has no facility for defining functions on sets or multisets.

There are two other kinds of set expressions that BLOG supports. One is an *explicit set expression*, in which the elements of the set are listed explicitly. For example, in the hurricane model (Figure 2), we could use the explicit set expression `{A, B}` rather than the implicit set expression `{City c}` on line 8. The last kind of set expression is a *cardinality expression*, which yields the size of an implicitly defined set. In the urn-and-balls example, we could use the cardinality expression `#{Ball b: TrueColor(b) = Blue}` to represent the number of blue balls in the urn.

We now formally define the syntax of BLOG set expressions. This definition uses the notion of a *scope* (a mapping from logical variables to types) from Chapter 2.

Definition 2. *A* well-formed set expression *of a BLOG model M in a scope β has one of the following forms:*

- *an* explicit set expression $\{t_1, \ldots, t_k\}$*, where $t_1, \ldots, t_k$ are well-formed terms of $\mathcal{L}_M$ in scope β;*

- *an* implicit set expression $\{\tau\ x\ :\ \varphi\}$*, where* τ *is a type in* M*,* x *is a logical variable symbol, and* φ *is a formula of* $\mathcal{L}_M$ *that is well-formed in the scope* $(\beta; x \mapsto \tau)$*;*

- *a* tuple multiset expression $\{t_1, \ldots, t_k$ `for` $\tau_1\ x_1, \ldots, \tau_n\ x_n\ :\ \varphi\}$*, where* $\tau_1, \ldots, \tau_n$ *are types in* M*,* $x_1, \ldots, x_n$ *are logical variable symbols,* $t_1, \ldots, t_k$ *are terms of* $\mathcal{L}_M$ *that are well-formed in the scope* $\beta' \equiv (\beta; x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_n)$*, and* φ *is a formula of* $\mathcal{L}_M$ *that is well-formed in* β'*;*

- *a* cardinality expression `#`s*, where* s *is a well-formed implicit set expression of* M *in* β*.*

If the condition φ in an implicit set expression or tuple multiset expression is simply `true`, then the condition and the colon that precedes it can be omitted. {`Ball b`} is an example of such an abbreviated expression.

A set expression may represent an infinite set, such as {`NaturalNum n`}, or {`Individual i : Female(i)`} in the recursive sexual reproduction model that we mentioned near the end of Section 2.6. In our Java implementation of BLOG, we do not have a general way to represent infinite sets or multisets so that they can be passed as arguments to elementary CPDs. Furthermore, since the number of infinite subsets of a given universe is uncountable, set expressions that can denote infinite sets cannot be thought of as discrete random variables. Thus, the elementary notion of conditional probability that we use throughout this thesis (see Chapter 2) does not allow us to define a CPD for a random variable given the value of such a set expression. To avoid these difficulties, we adopt the convention that a set expression that would otherwise denote an infinite set or multiset actually denotes `null`. Similarly, a cardinality expression involving an infinite set also denotes `null`.

Just as we defined the denotation of a term given a model structure and an assignment of values to logical variables (in Chapter 2), we can make a similar definition for set expressions.

Definition 3. *Let* ω *be a model structure of* $\mathcal{L}_M$*,* α *be an assignment that is valid in* ω*, and* s *be a set expression of* M *that is well-formed in* $\mathrm{domain}(\alpha)$*. Then the* denotation *of* s *in* ω *under* α*, denoted* $[s]^\omega_\alpha$*, is defined as follows:*

- *if* s *is an explicit set expression* $\{t_1, \ldots, t_k\}$*, then* $[s]^\omega_\alpha$ *is the set* $\{[t_i]^\omega_\alpha\ :\ i \in \{1, \ldots, k\}\}$*;*

- *if* s *is an implicit set expression* $\{\tau\ x\ :\ \varphi\}$*, then* $[s]^\omega_\alpha$ *is the set* $\{o \in [\tau]^\omega\ :\ \omega \models_{(\alpha,(x,\tau)\mapsto o)} \varphi\}$*, or* `null` *if that set is infinite;*

- *if* s *is a tuple multiset expression* $\{t_1, \ldots, t_k$ `for` $\tau_1\ x_1, \ldots, \tau_n\ x_n\ :\ \varphi\}$ *and we define* A *to be the set of valid assignments* $\alpha' = (\alpha; (x_1, \tau_1) \mapsto o_1, \ldots, (x_n, \tau_n) \mapsto$

$o_n)$ *in* ω *such that* $\omega \models_{\alpha'} \varphi$*, then* $[s]^\omega_\alpha$ *is the multiset* $\{|([t_1]^\omega_{\alpha'}, \ldots, [t_k]^\omega_{\alpha'}) : \alpha' \in A|\}$*, or* null *if* A *is infinite;*

- *if* s *is a cardinality expression* $\#s'$*, then* $[s]^\omega_\alpha$ *is* $|\,[s']^\omega_\alpha\,|$*, or* null *if* $[s']^\omega_\alpha = \mathsf{null}$*.*

Thus set expressions, like terms and formulas, can be evaluated to yield values in a possible world. The values of set expressions are always finite sets, finite multisets, natural numbers (for cardinality expressions), or null. From here on, we will refer to terms, formulas, and set expressions collectively as *BLOG expressions*. Also, for consistency, we will sometimes use the notation $[\varphi]^\omega_\alpha$ for the value of a formula φ in a model structure ω. That is, $[\varphi]^\omega_\alpha$ equals true if $\omega \models_\alpha \varphi$, and equals false otherwise.

2.9 Elementary CPDs

We have said so far that an elementary CPD is an instance of a Java class that implements a certain interface. But we would like our definition of a BLOG model to be independent of any particular implementation language. Thus, we give a more mathematical definition of an elementary CPD below. This definition enforces two assumptions, which can be stated informally as follows. If an elementary CPD is used in a model M and invoked with argument values $q_1, \ldots, q_n$:

1. the CPD can assign positive probability to an object o only if o is a guaranteed object in M or is contained in the argument tuple $(q_1, \ldots, q_n)$;

2. if h is a permutation of the non-guaranteed objects in M, then the probability that the CPD assigns to a non-guaranteed object o given $q_1, \ldots, q_n$ is the same as it assigns to $h(o)$ given $h(q_1), \ldots, h(q_n)$.

The first condition ensures that the CPD does not assign positive probability to objects that do not exist. The second condition asserts that non-guaranteed objects — for example, the balls in the urn-and-balls example — are treated interchangeably. Note that elementary CPDs are allowed to make distinctions among guaranteed objects: for instance, the tabular CPD for Damage in Figure 2 distinguishes between the two guaranteed PrepLevel objects that might be passed into it.

The way we stated these assumptions above ignores the fact that the arguments passed into an elementary CPD are not necessarily individual objects; they may be sets of objects, or multisets of tuples of objects, defined by a set expression. Thus, we need to be more careful in saying what it means for an expression value to "contain" an object, and what it means to apply a permutation h to an expression value.

Definition 4. *In a BLOG model* M*, an expression value* q contains *an object* o *if either* $q = o$*, or* $q \notin \mathrm{NonGuar}_M$ *and one of the following conditions holds:*

- *q is a set and for some $q' \in q$, q' contains o;*
- *q is a multiset and for some $q' \in q$, q' contains o;*
- *q is a tuple $(q_1, \ldots, q_k)$ and for some $i \in \{1, \ldots, k\}$, q_i contains o.*

Note that the recursion in this definition stops when it hits a non-guaranteed object, even if that non-guaranteed object happens to be a tuple (as discussed in the semantics section of the thesis, it is convenient to let non-guaranteed objects be tuples).

Definition 5. *Let M be a BLOG model, and let h be a permutation of* $\mathrm{NonGuar}_M$*: that is, a bijection from* $\mathrm{NonGuar}_M$ *to itself. The* extension of h to argument values *in M, denoted $\bar{h}$, is defined as follows on any expression value q:*

- *if $q \in \mathrm{NonGuar}_M$, then $\bar{h}(q) = h(q)$;*
- *otherwise:*
  - *if q is a set, then $\bar{h}(q) = \{\bar{h}(q') : q' \in q\}$;*
  - *if q is a multiset, then $\bar{h}(q) = \{|\bar{h}(q') : q' \in q|\}$;*
  - *if q is a tuple $(q_1, \ldots, q_k)$, then $\bar{h}(q) = (\bar{h}(q_1), \ldots, \bar{h}(q_k))$;*
  - *otherwise, $\bar{h}(q) = q$.*

We can now state a formal definition of an elementary CPD, including the restrictions that we stated less formally at the beginning of this section.

Definition 6. *An* elementary CPD *for a range set S in a discrete BLOG model M is a function c from the set of pairs $(o, (q_1, \ldots, q_k))$, where $o \in S$ and $(q_1, \ldots, q_k)$ is any tuple of expression values, to $[0, 1]$, such that:*

- *for each tuple of expression values $(q_1, \ldots, q_k)$, $\sum_{(o \in S)} c(o, (q_1, \ldots, q_k)) = 1$;*
- *if $c(o, (q_1, \ldots, q_k)) > 0$, then $o \in (\mathrm{Guar}_M(\tau) \cup \{\mathsf{null}\})$ or q_i contains o for some $i \in \{1, \ldots, k\}$;*
- *for any permutation h on* $\mathrm{NonGuar}_M$*,*

$$c(o, (q_1, \ldots, q_k)) = c(\bar{h}(o), (\bar{h}(q_1), \ldots, \bar{h}(q_k)))$$

for any object o and expression values $(q_1, \ldots, q_k)$.

Intuitively, the value $c(o, (q_1, \ldots, q_k))$ is to be interpreted as the conditional probability of the value o given the CPD arguments $q_1, \ldots, q_k$. As specified in previous sections, elementary CPDs in the dependency statement for a random function with return type r must have $\mathcal{O}_M(r) \cup \{\mathsf{null}\}$ as their range set; the range set for elementary CPDs in number statements must be $\mathbb{N}$. The semantics section of the thesis describes more formally the role that elementary CPDs play in the semantics of BLOG. In the full version of BLOG that includes Real and RkVector types, $c(o, (q_1, \ldots, q_k))$ is to be interpreted as a density at o rather than a probability; in this case, the value can range over $[0, \infty)$, not just $[0, 1]$.

Definition 6 requires an elementary CPD to accept any tuple of expression values $(q_1, \ldots, q_k)$ as arguments. In practice, an instance of the `TabularCPD` class expects a certain number of arguments, each being a single object; an instance of class `Uniform`, on the other hand, expects a single set as an argument. Currently, our interface for elementary CPDs does not include any facility for checking, when the model is loaded, that an argument list matches the CPD's expectations. During inference, if an elementary CPD gets an argument that it is not expecting, it can just assign probability 1 to null, and probability zero to all other values.

References

[Bar-Shalom and Fortmann, 1988] Y. Bar-Shalom and T. E. Fortmann. *Tracking and Data Association*. Academic Press, Boston, 1988.

[Lawrence *et al.*, 1999] S. Lawrence, C. L. Giles, and K. D. Bollacker. Autonomous citation matching. In *Proc. 3rd Int'l Conf. on Autonomous Agents*, pages 392–393, 1999.

[Pasula *et al.*, 2003] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *Advances in Neural Information Processing Systems 15*. MIT Press, Cambridge, MA, 2003.