1 BLOG: Probabilistic Models with Unknown Objects

Brian Milch
Computer Science Division
University of California at Berkeley, USA
milch@cs.berkeley.edu
http://www.cs.berkeley.edu/∼milch

Bhaskara Marthi
Computer Science Division
University of California at Berkeley, USA
bhaskara@cs.berkeley.edu
http://www.cs.berkeley.edu/∼bhaskara

Stuart Russell
Computer Science Division
University of California at Berkeley, USA
russell@cs.berkeley.edu
http://www.cs.berkeley.edu/∼russell

David Sontag
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, USA
dstontag@csail.mit.edu
http://people.csail.mit.edu/dsontag

Daniel L. Ong
Computer Science Division
University of California at Berkeley, USA
dlong@ocf.berkeley.edu
http://www.ocf.berkeley.edu/∼dlong

Andrey Kolobov
Computer Science Division
University of California at Berkeley, USA
karaya1@rambler.ru

1.1 Introduction

Human beings and AI systems must convert sensory input into some understanding of what is going on in the world around them. That is, they must make inferences about the objects and events that underlie their observations. No pre-specified list of objects is given; the agent must infer the existence of objects that were not known initially to exist.

In many AI systems, this problem of unknown objects is engineered away or resolved in a preprocessing step. However, there are important applications where the problem is unavoidable. *Population estimation*, for example, involves counting a population by sampling from it randomly and measuring how often the same object is resampled; this would be pointless if the set of objects were known in advance. *Record linkage*, a task undertaken by an industry of more than 300 companies, involves matching entries across multiple databases. These companies exist because of uncertainty about the mapping from observations to underlying objects. Finally, *multi-target tracking* systems perform *data association*, connecting, say, radar blips to hypothesized aircraft.

Probability models for such tasks are not new: Bayesian models for data association have been used since the 1960s [Sittler, 1964]. The models are written in English and mathematical notation and converted by hand into special-purpose code. This can result in inflexible models of limited expressiveness—for example, tracking systems assume independent trajectories with linear dynamics, and record linkage systems assume a naive Bayes model for fields in records. It seems natural, therefore, to seek a *formal language* in which to express probability models that allow for unknown objects.

Recent achievements in the field of probabilistic graphical models [Pearl, 1988] illustrate the benefits that can be expected from adopting a formal language: general-purpose inference algorithms, more sophisticated models, and techniques for automated model selection (structure learning). However, graphical models only describe fixed sets of random variables with fixed dependencies among them; they become awkward in scenarios with unknown objects. There has also been significant work on *first-order probabilistic languages* (FOPLs), which explicitly represent objects and the relations between them. We review some of this work in Section 1.7. However, most FOPLs make the assumptions of *unique names*, requiring that the symbols or terms of the language all refer to distinct objects, and *domain closure*, requiring that no objects exist besides the ones referred to by terms in the language. These assumptions are inappropriate for problems such as multi-target tracking, where we may want to reason about objects that are observed multiple times or that are not observed at all. Those FOPLs that do support unknown objects often do so in limited and *ad hoc* ways. In this chapter, we describe Bayesian logic (Blog) [Milch et al., 2005a], a new language that compactly and intuitively defines probability distributions over outcomes with varying sets of objects.

We begin in Section 1.2 with three example problems, each of which involves possible worlds with varying object sets and identity uncertainty. We show BLOG models for these problems and give initial, informal descriptions of the probability distributions that they define. Section 1.3 observes that the possible worlds in these scenarios are naturally viewed as model structures of *first-order logic*. It then defines precisely the set of possible worlds corresponding to a BLOG model. The key idea is a generative process that constructs a world by adding objects whose existence and properties depend on those of objects already created. In such a process, the existence of objects may be governed by many random variables, not just a single population size variable. Section 1.4 discusses exactly how a BLOG model specifies a probability distribution over possible worlds.

Section 1.5 solves a previously unnoticed "probabilistic Skolemization" problem: how to specify evidence about objects—such as radar blips—that one didn't know existed. Finally, Section 1.6 briefly discusses inference in unbounded outcome spaces, stating a sampling algorithm and a completeness theorem for a large class of BLOG models and giving experimental results on one particular model.

1.2 Examples

In this section we examine three typical scenarios with unknown objects—simplified versions of the population estimation, record linkage, and multitarget tracking problems mentioned above. In each case, we provide a short BLOG model that, when combined with a suitable inference engine, constitutes a working solution for the problem in question.

Example 1.1

An urn contains an unknown number of balls—say, a number chosen from a Poisson distribution. Balls are equally likely to be blue or green. We draw some balls from the urn, observing the color of each and replacing it. We cannot tell two identically colored balls apart; furthermore, observed colors are wrong with probability 0.2. How many balls are in the urn? Was the same ball drawn twice?

The BLOG model for this problem, shown in Figure 1.1, describes a stochastic process for generating worlds. The first 4 lines introduce the types of objects in these worlds—colors, balls, and draws—and the functions that can be applied to these objects. For each function, the model specifies a *type signature* in a syntax similar to that of C or Java. For instance, line 2 specifies that TrueColor is a random function that takes a single argument of type Ball and returns a value of type Color. Lines 5–7 specify what objects may exist in each world. In every world, there are exactly two distinct colors, blue and green, and there are exactly four draws. These are the *guaranteed* objects. On the other hand, different worlds have different numbers of balls, so the number of balls that exist is chosen from a prior—a Poisson with mean 6. Each ball is then given a color, as specified on line 8. Properties of the four draws are filled in by choosing a ball (line 9) and an observed color for that ball (lines

```
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]](TrueColor(BallDrawn(d)));
```

Figure 1.1 BLOG model for balls in an urn (Example 1.1) with four draws.

10–12). The probability of the generated world is the product of the probabilities of all the choices made.

Example 1.2

We have a collection of citations that refer to publications in a certain field. What publications and researchers exist, with what titles and names? Who wrote which publication, and to which publication does each citation refer? For simplicity, we just consider the title and author-name strings in these citations, which are subject to errors of various kinds, and we assume only single-author publications.

Figure 1.2 shows a BLOG model for this example, based on the model in [Pasula et al., 2003]. The BLOG model defines the following generative process. First, sample the total number of researchers from some distribution; then, for each researcher r, sample the number of publications by that researcher. Sample the researchers' names and publications' titles from appropriate prior distributions. Then, for each citation, sample the publication cited by choosing uniformly at random from the set of publications. Finally, generate the citation text with a "noisy" formatting distribution that allows for errors and abbreviations in the title and author names.

Example 1.3

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 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 some aircraft may not be detected. What aircraft exist, and what are their trajectories? Are there any aircraft that are not observed?

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

2   random String Name(Researcher);
3   random String Title(Publication);
4   random Publication PubCited(Citation);
5   random String Text(Citation);

6   origin Researcher Author(Publication);

7   guaranteed Citation Cite1, Cite2, Cite3, Cite4;

8   #Researcher ~ NumResearchersPrior();
9   #Publication(Author = r) ~ NumPubsPrior();

10  Name(r) ~ NamePrior();
11  Title(p) ~ TitlePrior();

12  PubCited(c) ~ Uniform({Publication p});

13  Text(c) ~ NoisyCitationGrammar(Title(PubCited(c)),
14                                 Name(Author(PubCited(c))));
```

Figure 1.2 BLOG model for Example 1.2 with four observed citations.

```
1   type Aircraft; type Blip;

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

4   nonrandom NaturalNum Pred(NaturalNum) = Predecessor;

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

7   #Aircraft ~ NumAircraftPrior();

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

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

13  ApparentPos(b)
14      if (Source(b) = null) then ~ FalseAlarmDistrib()
15      else ~ ObsCPD(State(Source(b), Time(b)));
```

Figure 1.3 BLOG model for Example 1.3.

The BLOG model for this scenario (Figure 1.3) 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 (position and velocity) 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 $\mathsf{Source}(b) = a$ and $\mathsf{Time}(b) = t$. Whether a blip is generated or not depends on the state of the aircraft—thus the number of objects in the world depends on certain objects' attributes. Also, at each step t, generate some false alarm blips b' with $\mathsf{Time}(b') = t$ and $\mathsf{Source}(b') = \mathsf{null}$. 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).

1.3 Syntax and Semantics: Possible Worlds

1.3.1 Outcomes as first-order model structures

The possible outcomes for Examples 1 through 3 are structures containing many related objects, with the set of objects and the relations among them varying from outcome to outcome. We will treat these outcomes formally as *model structures* of *first-order logic*. A model structure provides interpretations for the symbols of a first-order language; each sentence of the first-order language can be evaluated to yield a truth value in each model structure.

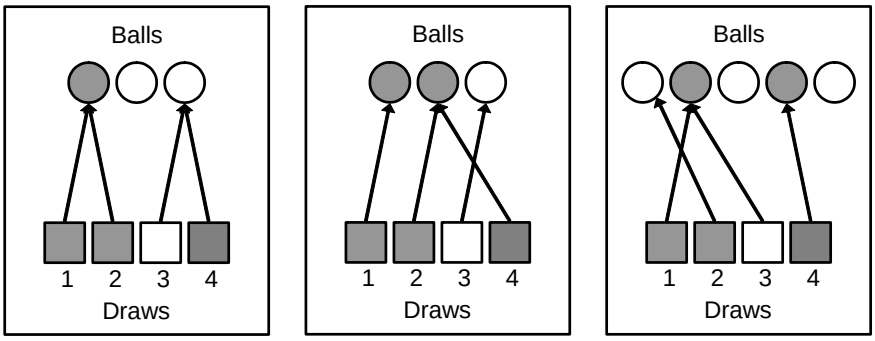
In Example 1.1, the language has function symbols such as $\mathsf{TrueColor}(b)$ for the true color of ball b; $\mathsf{BallDrawn}(d)$ for the ball drawn on draw d; and $\mathsf{Draw1}$ for the first draw. (Usually, first-order languages are described as having predicate, function, and constant symbols. For conciseness, we view all symbols as function symbols; predicates are just functions that return a Boolean value, and constants are just zero-ary functions.) To eliminate meaningless random variables, we use *typed* logical languages. Each BLOG model uses a language with a particular set of types, such as Ball and Draw. BLOG also has some built-in types that are available in all models, namely $\mathsf{Boolean}$, $\mathsf{NaturalNum}$, $\mathsf{Integer}$, String, Real, and $\mathsf{R}k\mathsf{Vector}$ (for each $k \geq 2$). Each function symbol f has a *type signature* $(\tau_0, \ldots, \tau_k)$, where τ_0 is the return type of f and $\tau_1, \ldots, \tau_k$ are the argument types. The type $\mathsf{Boolean}$ receives special syntactic treatment: if the return type of a function f is $\mathsf{Boolean}$, then terms of the form $f(t_1, \ldots, t_k)$ constitute atomic formulas, which can be combined using logical operators and placed inside quantifiers.

The logical languages used in BLOG are also *free*: a function is not required to apply to all tuples of arguments, even if they are appropriately typed [Lambert, 1998]. For instance, in Example 1.3, the function Source usually maps blips to aircraft, but it is not applicable if the blip is a false detection. We adopt the convention that when a function is not applicable to some arguments, it returns the special value null. Any function that receives null as an argument also returns null, and an atomic formula that evaluates to null is treated as false.

The truth of any first-order sentence is determined by a *model structure* for the

corresponding language. A model structure specifies the *extension* of each type and the *interpretation* for each function symbol:

Definition 1.1
A model structure ω of a typed, free, first-order language consists of an extension $[\tau]^\omega$ for each type τ, which may be an arbitrary set, and an interpretation $[f]^\omega$ for each function symbol f. If f has return type τ_0 and argument types $\tau_1, \ldots, \tau_k$, then $[f]^\omega$ is a function from $[\tau_1]^\omega \times \cdots \times [\tau_k]^\omega$ to $[\tau_0]^\omega \cup \{\mathsf{null}\}$.



(a) (b) (c)

Figure 1.4 Three model structures for the language of Figure 1.1. Shaded circles represent balls that are blue; shaded squares represent draws where the drawn ball appeared blue (unshaded means green). Arrows represent the BallDrawn function from draws to balls.

Three model structures for the language used in Figure 1.1 are shown in Figure 1.4. Identity uncertainty arises because $[\mathsf{BallDrawn}]^\omega(\mathsf{Draw1})$ might be equal to $[\mathsf{BallDrawn}]^\omega(\mathsf{Draw2})$ in one structure (such as Figure 1.4(a)) but not another (such as Figure 1.4(b)). The set of balls, $[\mathsf{Ball}]^\omega$, can also vary between structures, as Figure 1.4 illustrates. The purpose of a BLOG model is to define a probability distribution over such structures. Because any sentence can be evaluated as true or false in each model structure, a distribution over model structures implicitly defines the probability that φ is true for each sentence φ in the logical language.

1.3.2 Outcomes with fixed object sets

We begin our formal discussion of BLOG semantics by considering the relatively simple case of models with fixed sets of objects. BLOG models for fixed object sets have five kinds of statements. A *type declaration*, such as the two statements on line 1 of Figure 1.3, introduces a type. A *random function declaration*, such as line 2 of Figure 1.3, specifies the type signature for a function symbol whose values will be chosen randomly in the generative process. A *nonrandom function definition*,

such as the one on line 4 of Figure 1.3, introduces a function whose interpretation is fixed in all possible worlds. In our implementation, the interpretation is given by a Java class (`Predecessor` in this example). A *guaranteed object statement*, such as line 5 in Figure 1.1, introduces and names some distinct objects that exist in all possible worlds. For the built-in types, the obvious sets of guaranteed objects and constant symbols are predefined. The set of guaranteed objects of type τ in BLOG model M is denoted $G_M(\tau)$. Finally, for each random function symbol, a BLOG model includes a *dependency statement* specifying how values are chosen for that function. We postpone further discussion of dependency statements to Section 1.4.

The first four kinds of statements listed above define a particular typed first-order language $\mathcal{L}_M$ for a model M. The set of *possible worlds* of M, denoted Ω_M, consists of those model structures of $\mathcal{L}_M$ where the extension of each type τ is $G_M(\tau)$, and all nonrandom function symbols (including guaranteed constants) have their given interpretations.

For each random function f and tuple of appropriately typed guaranteed objects $o_1, \ldots, o_k$, we can define a random variable (RV) $f[o_1, \ldots, o_k](\omega) \triangleq [f]^{\omega}(o_1, \ldots, o_k)$. For instance, in a simplified version of Example 1.1 where the urn contains a known set of balls $\{\mathsf{Ball1}, \ldots, \mathsf{Ball8}\}$ and we make four draws, the random variables are $\mathsf{TrueColor}[\mathsf{Ball1}], \ldots, \mathsf{TrueColor}[\mathsf{Ball8}]$, $\mathsf{BallDrawn}[\mathsf{Draw1}], \ldots, \mathsf{BallDrawn}[\mathsf{Draw4}]$, and $\mathsf{ObsColor}[\mathsf{Draw1}], \ldots, \mathsf{ObsColor}[\mathsf{Draw4}]$. The possible worlds are in one-to-one correspondence with full instantiations of these basic RVs. Thus, a joint distribution for the basic RVs defines a distribution over possible worlds.

1.3.3 Unknown objects

In general, a BLOG model defines a generative process in which objects are added iteratively to a world. To describe such processes, we first introduce *origin function declarations*[1], such as lines 5–6 of Figure 1.3. Unlike other functions, origin functions such as Source or Time have their values set when an object is added. An origin function must take a single argument of some type τ (namely Blip in the example); it is then called a τ-origin function.

Generative steps that add objects to the world are described by *number statements*, such as line 11 of Figure 1.3:

```
#Blip(Source = a, Time = t) ~ DetectionCPD(State(a, t));
```

This statement says that for each aircraft a and time step t, the process adds some number of blips, and each of these added blips b has the property that $\mathsf{Source}(b) = a$ and $\mathsf{Time}(b) = t$. In general, the beginning of a number statement has the form:

$\#\tau(g_1 = x_1, \ldots, g_k = x_k)$

1. In [Milch et al., 2005a] we used the term "generating function", but we have now adopted the term "origin function" because it seems clearer.

where τ is a type, $g_1, \ldots, g_k$ are τ-origin functions, and $x_1, \ldots, x_k$ are logical variables. (For types that are generated *ab initio* with no origin functions, the empty parentheses are omitted, as in Figure 1.1.) The inclusion of a number statement means that for each appropriately typed tuple of objects $o_1, \ldots, o_k$, the generative process adds some random number (possibly zero) of objects q of type τ such that $[g_i]^\omega(q) = o_i$ for $i = 1, \ldots, k$. Note that the types of the generating objects $o_1, \ldots, o_k$ are the return types of $g_1, \ldots, g_k$.

Object generation can even 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();
```

We can also view number statements more declaratively:

Definition 1.2
Let ω be a model structure of $\mathcal{L}_M$, and consider a number statement for type τ with origin functions $g_1, \ldots, g_k$. An object $q \in [\tau]^\omega$ *satisfies* this number statement applied to $o_1, \ldots, o_k$ in ω if $[g_i]^\omega(q) = o_i$ for $i = 1, \ldots, k$, and $[g]^\omega(q) = \mathsf{null}$ for all other τ-origin functions g.

Note that if a number statement for type τ omits one of the τ-origin functions, then this function takes on the value null for all objects satisfying that number statement. For instance, Source is null for objects satisfying the false-detection number statement on line 12 of Figure 1.3:

```
#Blip(Time = t) ~ NumFalseAlarmsPrior();
```

Also, a BLOG model cannot contain two number statements with the same set of origin functions. This ensures that, in any given model structure, each object o has exactly one generation history, which can be found by tracing back the origin functions on o.

The set of possible worlds Ω_M is the set of model structures that can be constructed by M's generative process. To complete the picture, we must explain not only *how many* objects are added on each step, but also *what* these objects are. It turns out to be convenient to define the generated objects as follows: when a number statement with type τ and origin functions $g_1, \ldots, g_k$ is applied to generating objects $o_1, \ldots, o_k$, the generated objects are tuples $\{(\tau, (g_1, o_1), \ldots, (g_k, o_k), n) : n = 1, \ldots, N\}$, where N is the number of objects generated. Thus in Example 1.3, the aircraft are pairs $(\mathsf{Aircraft}, 1)$, $(\mathsf{Aircraft}, 2)$, etc., and the blips generated by aircraft are nested tuples such as $(\mathsf{Blip}, (\mathsf{Source}, (\mathsf{Aircraft}, 2)), (\mathsf{Time}, 8), 1)$. The tuple encodes the object's generation history; of course, it is purely internal to the semantics and remains invisible to the user.

Definition 1.3
The *universe* of a type τ in a BLOG model M, denoted $U_M(\tau)$, consists of the guaranteed objects of type τ as well as all nested tuples of type τ that can be generated from the guaranteed objects through finitely many recursive applications of number statements.

As the following definition stipulates, in each possible world the extension of τ is some subset of $U_M(\tau)$.

Definition 1.4
For a BLOG model M, the set of possible worlds Ω_M is the set of model structures ω of $\mathcal{L}_M$ such that:

1. for each type τ, $G_M(\tau) \subseteq [\tau]^\omega \subseteq U_M(\tau)$;
2. nonrandom functions have the specified interpretations;
3. for each number statement in M with type τ and origin functions $g_1, \ldots, g_k$, and each appropriately typed tuple of generating objects $(o_1, \ldots, o_k)$ in ω, the set of objects in $[\tau]^\omega$ that satisfy this number statement applied to these generating objects is $\{(\tau, (g_1, o_1), \ldots, (g_k, o_k), n) : n = 1, \ldots, N\}$ for some natural number N;
4. for every type τ, each element of $[\tau]^\omega$ satisfies some number statement applied to some objects in ω.

Note that by part 3 of this definition, the number of objects generated by any given application of a number statement in world ω is a finite number N. However, a world can still contain infinitely many non-guaranteed objects if some number statements are applied recursively: then the world may contain tuples that are nested to depths $1, 2, 3, \ldots$, with no upper bound. Infinitely many objects can also result if number statements are triggered for every natural number, like the statements that generate radar blips in Example 1.3.

With a fixed set of objects, it was easy to define a set of basic RVs such that a full instantiation of the basic RVs uniquely identified a possible world. To achieve the same effect with unknown objects, we need two kinds of basic RVs:

Definition 1.5
For a BLOG model M, the set $\mathbf{V}_M$ of *basic random variables* consists of:

- for each random function f with type signature $(\tau_0, \ldots, \tau_k)$ and each tuple of objects $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$, a *function application RV* $f[o_1, \ldots, o_k](\omega)$ that is equal to $[f]^\omega(o_1, \ldots, o_k)$ if $o_1, \ldots, o_k$ all exist in ω, and null otherwise;
- for each number statement with type τ and origin functions $g_1, \ldots, g_k$ that have return types $\tau_1, \ldots, \tau_k$, and each tuple of objects $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$, a *number RV* $\#\tau[g_1 = o_1, \ldots, g_k = o_k](\omega)$ equal to the number of objects that satisfy this number statement applied to $o_1, \ldots, o_k$ in ω.

Intuitively, each step in the generative world-construction process determines the

value of a basic variable. The crucial result about basic RVs is the following:

Proposition 1.6
For any BLOG model M and any complete instantiation of $\mathbf{V}_M$, there is at most one model structure in Ω_M consistent with this instantiation.

Some instantiations of $\mathbf{V}_M$ do not correspond to any possible world: for example, an instantiation for the urn-and-balls example where $\#\mathsf{Ball}[] = 2$, but $\mathsf{TrueColor}[(\mathsf{Ball}, 7)]$ is not null. Instantiations of $\mathbf{V}_M$ that correspond to a world are called *achievable*. Thus, to define a probability distribution over Ω_M, it suffices to define a joint distribution over the achievable instantiations of $\mathbf{V}_M$.

Now that we have seen this technical development, we can say more about the need to represent objects as tuples that encode generation histories. Equating objects with tuples might seem unnecessarily complicated, but it becomes very helpful when we define a Bayes net over the basic RVs (which we do in Section 1.4.2). For instance, in the aircraft tracking example, the parent of $\mathsf{ApparentPos}[(\mathsf{Blip}, (\mathsf{Source}, (\mathsf{Aircraft}, 2)), (\mathsf{Time}, 8), 1)]$ is $\mathsf{State}[(\mathsf{Aircraft}, 2), 8]$. It might seem more elegant to assign numbers to objects as they are generated, so that the extension of each type in each possible world would be simply a prefix of the natural numbers. Specifically, we could number the aircraft arbitrarily, and then number the radar blips lexicographically by aircraft and time step. Then we would have basic RVs such as $\mathsf{ApparentPos}[23]$, representing the apparent aircraft position for blip 23. But blip 23 could be generated by any aircraft at any time step. In fact, the parents of $\mathsf{ApparentPos}[23]$ would have to include all the $\#\mathsf{Blip}$ and State variables in the model. So defining objects as tuples yields a much simpler Bayes net.

1.4 Syntax and Semantics: Probabilities

1.4.1 Dependency statements

Dependency and number statements specify exactly how the steps are carried out in our generative process. Consider the dependency statement for $\mathsf{State}(a, t)$ from Figure 1.3:

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

This statement is applied for every basic RV of the form $\mathsf{State}[a, t]$ where $a \in U_M(\mathsf{Aircraft})$ and $t \in \mathbb{N}$. 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 $\mathsf{R6Vector}$ (the return type of State).

In our implementation, elementary CPDs are Java classes with a method `getProb` that returns the probability of a particular value given a list of CPD arguments, and a method `sampleVal` that samples a value given the CPD arguments.

A dependency statement begins with a function symbol f and a tuple of logical variables $x_1, \ldots, x_k$ representing the arguments to this function. In a number statement, the variables $x_1, \ldots, x_k$ represent the generating objects. In either case, the rest of the statement consists of a sequence of *clauses*. When the statement is not abbreviated, the syntax for the first clause is:

`if` *cond* `then` $\sim$ *elem-cpd*(*arg1*, ..., *argN*)

The *cond* portion is a formula of the first-order logical language $\mathcal{L}_M$ (containing no free variables other than $x_1, \ldots, x_k$) specifying the condition under which this clause should be used to sample a value for a basic RV. More precisely, if the possible world constructed so far is ω, then the applicable clause is the *first* one whose condition is satisfied in ω (assuming for the moment that ω is complete enough to determine the truth values of the conditions). If no clause's condition is satisfied, or if the basic RV refers to objects that do not exist in ω, then the value is set by default to false for Boolean functions, null for other functions, and zero for number variables. If the condition in a clause is just "`true`", then the whole string "`if true then`" may be omitted.

In the applicable clause, each CPD argument is evaluated in ω. The resulting values are then passed to the elementary CPD. In the simplest case, the arguments are terms or formulas of $\mathcal{L}_M$, such as $\mathsf{State}(a, \mathsf{Pred}(t))$. An argument can also be a *set expression* of the form $\{\tau\ y : \varphi\}$, where τ is a type, y is a logical variable, and φ is a formula. The value of such an expression is the set of objects $o \in [\tau]^\omega$ such that ω satisfies φ with y bound to o. If the formula φ is just true it can be omitted: this is the case on line 9 of Figure 1.1, where we just see the expression {`Ball b`}. BLOG also includes other kinds of arguments to allow counting the number of elements in a set, aggregating a multiset of values, or passing in a set of pairs (o, w) where the o's are objects and the w's are non-uniform sampling weights.

We require that the elementary CPDs obey two rules related to non-guaranteed objects. First, if a CPD is defining a distribution over non-guaranteed objects (*e.g.*, the `Uniform` CPD on line 9 of Figure 1.1), it should never assign positive probability to objects that do not exist in the partially completed world ω. To ensure this, we allow an elementary CPD to assign positive probability to a non-guaranteed object only if the object was passed in as part of a CPD argument (in Figure 1.1, {`Ball b`} is passed in). Second, an elementary CPD cannot "peek" at the tuple representations of objects that are passed in: it must be invariant to permutations of the non-guaranteed objects.

1.4.2 Declarative semantics

So far we have explained BLOG semantics procedurally, in terms of a generative process. To facilitate both knowledge engineering and the development of learning

algorithms, we would like to have declarative semantics. The standard approach — which is used in most existing FOPLs — is to say that a BLOG model defines a certain Bayesian network (BN) over the basic RVs. In this section we discuss how that approach needs to be modified for BLOG.

We will write σ to denote an instantiation of a set of RVs $\text{vars}(\sigma)$, and σ_X to denote the value that σ assigns to X. If a BN is finite, then the probability it assigns to each complete instantiation σ is $P(\sigma) = \prod_{X \in \text{vars}(\sigma)} p_X(\sigma_X | \sigma_{\text{Pa}(X)})$, where p_X is the CPD for X and $\sigma_{\text{Pa}(X)}$ is σ restricted to the parents of X. In an infinite BN, we can write a similar expression for each *finite* instantiation σ that is closed under the parent relation (that is, $X \in \text{vars}(\sigma)$ implies $\text{Pa}(X) \subseteq \text{vars}(\sigma)$). If the BN is acyclic and each variable has finitely many ancestors, then these probability assignments define a unique distribution [Kersting and De Raedt, 2001].

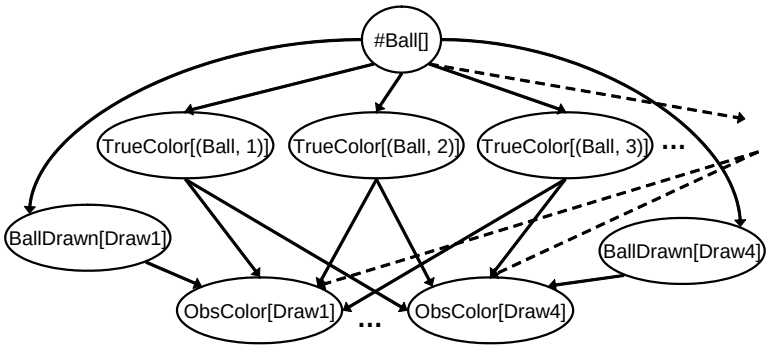


Figure 1.5 Bayes net for the BLOG model in Figure 1.1. The ellipses and dashed arrows indicate that there are infinitely many $\mathsf{TrueColor}[b]$ nodes.

The difficulty is that in the BN corresponding to a BLOG model, variables often have infinite parent sets. For instance, the BN for Example 1.1 (shown partially in Figure 1.5) has an infinite number of basic RVs of the form $\mathsf{TrueColor}[b]$: if it had only a finite number N of these RVs, it could not represent outcomes with more than N balls. Furthermore, each of these $\mathsf{TrueColor}[b]$ RVs is a parent of each $\mathsf{ObsColor}[d]$ RV, since if $\mathsf{BallDrawn}[d]$ happens to be b, then the observed color on draw d depends directly on the color of ball b. So the $\mathsf{ObsColor}[d]$ nodes have infinitely many parents. In such a model, assigning probabilities to finite instantiations that are closed under the parent relation does not define a unique distribution: in particular, it tells us nothing about the $\mathsf{ObsColor}[d]$ variables.

We required instantiations to be closed under the parent relation so that the factors $p_X(\sigma_X | \sigma_{\text{Pa}(X)})$ would be well-defined. But we may not need the values of *all* of X's parents in order to determine the conditional distribution for X. For instance, knowing $\mathsf{BallDrawn}[d] = (\mathsf{Ball}, 13)$ and $\mathsf{TrueColor}[(\mathsf{Ball}, 13)] = \mathsf{Blue}$ is sufficient to determine the distribution for $\mathsf{ObsColor}[d]$: the colors of all the other balls are irrelevant in this context. We can read off this context-specific independence from the dependency statement for $\mathsf{ObsColor}$ in Figure 1.1 by noting that the in-

stantiation ($\mathsf{BallDrawn}[d] = (\mathsf{Ball}, 13)$, $\mathsf{TrueColor}[(\mathsf{Ball}, 13)] = \mathsf{Blue}$) determines the value of the sole CPD argument $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$. We say this instantiation *supports* the variable $\mathsf{ObsColor}[d]$ (see [Milch et al., 2005b]).

Definition 1.7
An instantiation σ *supports* a basic RV V of the form $f[o_1, \ldots, o_k]$ or $\#\tau[g_1 = o_1, \ldots, g_k = o_k]$ if all possible worlds consistent with σ agree on (1) whether all the objects $o_1, \ldots, o_k$ exist, and, if so, on (2) the applicable clause in the dependency or number statement for V and the values for the CPD arguments in that clause.

Note that some RVs, such as $\#\mathsf{Ball}[]$ in Example 1.1, are supported by the empty instantiation. We can now generalize the notion of being closed under the parent relation.

Definition 1.8
A finite instantiation σ is *self-supporting* if its instantiated variables can be numbered $X_1, \ldots, X_N$ such that for each $n \leq N$, the restriction of σ to $\{X_1, \ldots, X_{n-1}\}$ supports X_n.

This definition lets us give semantics to BLOG models in a way that is meaningful even when the corresponding BNs contain infinite parent sets. We will write $p_V(v \mid \sigma)$ for the probability that V's dependency or number statement assigns to the value v, given an instantiation σ that supports V.

Definition 1.9
A distribution P over Ω_M *satisfies* a BLOG model M if for every finite, self-supporting instantiation σ with $\mathrm{vars}(\sigma) \subseteq \mathbf{V}_M$:

$$P(\Omega_\sigma) = \prod_{n=1}^{N} p_{X_n}(\sigma_{X_n} \mid \sigma_{\{X_1,\ldots,X_{n-1}\}}) \tag{1.1}$$

where Ω_σ is the set of possible worlds consistent with σ and $X_1, \ldots, X_N$ is a numbering of σ as in Definition 1.8.

A BLOG model is *well-defined* if there is exactly one probability distribution that satisfies it. Recall that a BN is well-defined if it is acyclic and each variable has a finite set of ancestors. Another way of saying this is that each variable can be "reached" by enumerating its ancestors in a finite, topologically ordered list. The well-definedness criterion for BLOG is similar, but deals with finite, self-supporting instantiations rather than finite, topologically ordered lists of variables. Because we are dealing with instantiations rather than variables, we need to make sure that they cover all possible worlds in addition to covering all basic variables.

Theorem 1.10
Let M be a BLOG model. Suppose that $\mathbf{V}_M$ is at most countably infinite,[2] and for each $V \in \mathbf{V}_M$ and $\omega \in \Omega_M$, there is a self-supporting instantiation that agrees with ω and includes V. Then M is well-defined.

Proof We provide only a sketch of the proof here, deferring the full version to a more technical paper. First, since $\mathbf{V}_M$ is at most countably infinite, we can impose an arbitrary numbering (a bijection with some prefix of the natural numbers) on $\mathbf{V}_M$. This numbering is "global" in the sense that it does not depend on the instantiation of the random variables. Now, we define a sequence of auxiliary random variables $\{Y_n : 0 \leq n < |\mathbf{V}_M|\}$ on Ω_M as follows. Let $Y_0(\omega) = X(\omega)$ where X is the first basic RV in the global ordering that is supported by the empty instantiation. For $n \geq 1$, let $\sigma_n(\omega)$ be the instantiation $(Y_0 = Y_0(\omega), \ldots, Y_{n-1} = Y_{n-1}(\omega))$. Then let $Y_n(\omega) = Z(\omega)$ where Z is the first basic RV in the global ordering that is supported by $\sigma_n(\omega)$, but has not already been used to define $Y_m(\omega)$ for any $m < n$. The important property of the sequence $\{Y_n\}$ is that any instantiation of $Y_0, \ldots, Y_{n-1}$ determines the CPD for Y_n. In other words, if we define our model in terms of $\{Y_n\}$, we get a standard BN in which each variable has finitely many ancestors.

However, we must show that this sequence $\{Y_n\}$ is well-defined. Specifically, we must show that for every $n < |\mathbf{V}_M|$ and every $\omega \in \Omega_M$, there exists a basic RV Z that is supported by $\sigma_n(\omega)$ and has not already been used to define $Y_m(\omega)$ for some $m < n$. This can be shown using the premise that for every $V \in \mathbf{V}_M$, there is a self-supporting instantiation consistent with ω that contains V.

We can use standard results from probability theory to show that there is a unique probability distribution over full instantiations of $\{Y_n\}$ such that each Y_n has the specified conditional distribution given all its predecessors. It remains to show that this distribution over instantiations corresponds to a unique distribution on Ω_M. First, we must show that each full instantiation of $\{Y_n\}$ corresponds to at most one possible world: this follows from Proposition 1.6, plus the fact that a full instantiation of $\{Y_n\}$ determines all the basic RVs. Second, we can show that the probability distribution we have defined over $\{Y_n\}$ is concentrated on instantiations that actually correspond to possible worlds — not instantiations that give RVs values of the wrong type, or give RVs non-null values in contexts where they must be null.

Finally, we need to check that this unique distribution on Ω_M indeed satisfies M. For finite, self-supporting instantiations σ that correspond to the auxiliary instantiations $\sigma_n(\omega)$ used in defining $\{Y_n\}$, the constraint is satisfied by construction. All other finite, self-supporting instantiations can be expressed as disjunctions of those "core" instantiations. From these observations, it is possible to show that (1.1) is satisfied for all finite, self-supporting instantiations. ■

2. This is satisfied if the Real and R*k*Vector types are not arguments to random functions or return types of gorigin functions.

To check that the criterion of Theorem 1.10 holds for a particular example, we need to consider each basic RV. In Example 1.1, the number RV for balls is supported by the empty instantiation, so in every world it is part of a self-supporting instantiation of size one. Each TrueColor $[b]$ RV depends only on whether its argument exists, so these variables participate in self-supporting instantiations of size two. Similarly, each BallDrawn variable depends only on what balls exist. To sample an ObsColor $[d]$ variable, we need to know BallDrawn $[d]$ and TrueColor [BallDrawn $[d]$], so these variables are in self-supporting instantiations of size four. Similar arguments can be made for Examples 1.2 and 1.3. Of course, we would like to have an algorithm for checking whether a BLOG model is well-defined; the criteria given in Theorem 1.12 in Section 1.6.2 are a first step in this direction.

1.5 Evidence and Queries

Because a well-defined BLOG model M defines a distribution over model structures, we can use arbitrary sentences of $\mathcal{L}_M$ as evidence and queries. But sometimes such sentences are not enough. In Example 1.3, the user observes radar blips, which are not referred to by any terms in the language. The user could assert evidence about the blips using existential quantifiers, but then how could he make a query of the form, "Did *this* blip come from the same aircraft as *that* blip?"

A natural solution is to allow the user to extend the language when evidence arrives, adding constant symbols to refer to observed objects. In many cases, the user observes some new objects, introduces some new symbols, and assigns the symbols to the objects in an uninformative order. To handle such cases, BLOG includes a special macro. For instance, given 4 radar blips at time 8, one can assert:

```
{Blip r: Time(r) = 8} = {Blip1, Blip2, Blip3, Blip4};
```

This asserts that there are exactly 4 radar blips at time 8, and introduces new constants Blip1, ..., Blip4 in one-to-one correspondence with those blips.

Formally, the macro augments the model with dependency statements for the new symbols. The statements implement sampling without replacement; for our example, we have

```
Blip1 ~ Uniform({Blip r : (Time(r) = 8)});
Blip2 ~ Uniform({Blip r : (Time(r) = 8) & (Blip1 != r)});
```

and so on. Once the model has been extended this way, the user can make assertions about the apparent positions of Blip1, Blip2, etc., and then use these symbols in queries.

These new constants resemble Skolem constants, but conditioning on assertions about the new constants is *not* the same as conditioning on an existential sentence. For example, suppose you go into a new wine shop, pick up a bottle at random, and observe that it costs $40. This scenario is correctly modeled by introducing a new

constant Bottle1 with a `Uniform` CPD. Then observing that Bottle1 costs at least $40 suggests that this is a fancy wine shop. On the other hand, the mere *existence* of a $40+ bottle in the shop does not suggest this, because almost every shop has *some* bottle at over $40.

1.6 Inference

Because the set of basic RVs of a BLOG model can be infinite, it is not obvious that inference for well-defined BLOG models is even decidable. However, the generative process intuition suggests a rejection sampling algorithm. We present this algorithm not because it is particularly efficient, but because it demonstrates the decidability of inference for a large class of BLOG models (see Theorem 1.12 below) and illustrates several issues that any BLOG inference algorithm must deal with. At the end of this section, we present experimental results from a somewhat more efficient likelihood weighting algorithm.

1.6.1 Rejection sampling

Suppose we are given a partial instantiation e as evidence, and a query variable Q. To generate each sample, our rejection sampling algorithm starts with an empty instantiation σ. Then it repeats the following steps: enumerate the basic RVs in a fixed order[3] until we reach the first RV V that is supported by σ but not already instantiated in σ; sample a value v for V according to V's dependency statement; and augment σ with the assignment $V = v$. The process continues until all the query and evidence variables have been sampled. If the sample is consistent with the evidence e, then the program increments a counter N_q, where q is the sampled value of Q. Otherwise, it rejects this sample. After N accepted samples, the estimate of $P(Q = q \mid e)$ is N_q/N.

This algorithm requires a subroutine that determines whether a partial instantiation σ supports a basic RV V, and if so, returns a sample from V's conditional distribution. For a basic RV V of the form $f[o_1, \ldots, o_k]$ or $\#\tau[g_1 = o_1, \ldots, g_k = o_k]$, the subroutine begins by checking the values of the relevant number variables in σ to determine whether all of $o_1, \ldots, o_k$ exist. If some of these number variables are not instantiated, then σ does not support V. If some of $o_1, \ldots, o_k$ do not exist, the subroutine returns the default value for V. If they do all exist, the subroutine follows the semantics for dependency statements discussed in Section 1.4.1. First, it iterates over the clauses in the dependency (or number) statement until it reaches

3. Each basic RV $f[o_1, \ldots, o_k]$ or $\#\tau[g_1 = o_1, \ldots, g_k = o_k]$ can be assigned a "depth" which is the maximum of the depths of nested tuples and the magnitudes of integers among its arguments $o_1, \ldots, o_k$. The number of RVs at each given depth is finite. Thus, we can enumerate first the RVs at depth 0, then those at depth 1, depth 2, etc.

a clause whose condition is either undetermined or determined to be true given σ (if all the conditions are determined to be false, then it returns the default value for V). If the condition is undetermined, then σ does not support V. If it is determined to be true, then the subroutine evaluates each of the CPD arguments in this clause. If σ determines the values of all the arguments, then the subroutine samples a value for V by passing those values to the `sampleVal` method of this clause's elementary CPD. Otherwise, σ does not support V.

To evaluate terms and quantifier-free formulas, we use a straightforward recursive algorithm. The base case looks up the value of a particular function application RV in σ; if this RV is not instantiated, the algorithm returns "undetermined". To evaluate a formula, we evaluate its subformulas in order from left to right. We stop when we hit an undetermined subformula or when the value of the whole formula is determined. For example, to evaluate $\alpha \vee \beta$, we first evaluate α. If α is undetermined, we return "undetermined"; if α is true, we return `true`, and if α is false, we go on to evaluate β.[4]

It is more complicated to evaluate set expressions such as {`Blip r: Time(r) = 8`}, which can be used as CPD arguments. A naive algorithm for evaluating this expression would first enumerate all the objects of type Blip (which would require certain number variables to be instantiated), then select the blips r that satisfy $\mathsf{Time}(r) = 8$. But Figure 1.3 specifies that there may exist some blips for each aircraft a and each natural number t: since there are infinitely many natural numbers, some worlds contain infinitely many blips. Fortunately, the number of blips r with $\mathsf{Time}(r) = 8$ is necessarily finite: in every world there are a finite number of aircraft, and each one generates a finite number of blips at time 8. We have an algorithm that scans the formula within a set expression for *origin function restrictions* such as $\mathsf{Time}(r) = 8$, and uses them to avoid enumerating infinite sets when possible. These restrictions may be either equality constraints, or inequalities that define a bounded set of natural numbers, such as $\mathsf{Time}(r) < 12$. A similar method is used for evaluating quantified formulas.

1.6.2 Termination criteria

In order to generate each sample, the algorithm above repeatedly instantiates the first variable that is supported but not yet instantiated, until it instantiates all the query and evidence variables. When can we be sure that this will take a finite amount of time? The first way this process could fail to terminate is if it goes into an infinite loop while checking whether a particular variable is supported. This happens if the program ends up enumerating an infinite set while evaluating a set expression or quantified formula. We can avoid this by ensuring that all such

4. This left-to-right evaluation scheme does not always detect that a formula is determined: for instance, on $\alpha \vee \beta$, it returns "undetermined" if α is undetermined but β is true—even though $\alpha \vee \beta$ must be true in this case.

expressions in the BLOG model are finite once origin function restrictions are taken into account.

The sample generator also fails to terminate if it never constructs an instantiation that supports a particular query or evidence variable. To see how this can happen, consider calling the subroutine described above to sample a variable V. If V is not supported, the subroutine will realize this when it encounters a variable U that is relevant but not instantiated. Now consider a graph over basic variables where we draw an edge from U to V when the evaluation process for V hits U in this way. If a variable is never supported, then it must be part of a cycle in this graph, or part of a receding chain of variables $V_1 \leftarrow V_2 \leftarrow \cdots$ that is extended infinitely.

The graph constructed in this way varies from sample to sample: for instance, sometimes the evaluation process for $\mathsf{ObsColor}[d]$ will hit $\mathsf{TrueColor}[(\mathsf{Ball}, 7)]$, and sometimes it will hit $\mathsf{TrueColor}[(\mathsf{Ball}, 13)]$. However, we can rule out cycles and infinite receding chains in all these graphs by considering a more abstract graph over function symbols and types (along the same lines as the *dependency graph* of [Koller and Pfeffer, 1998, Friedman et al., 1999]).

Definition 1.11
The *symbol graph* for a BLOG model M is a directed graph whose nodes are the types and random function symbols of M, where the parents of a type τ or function symbol f are:

- the random function symbols that occur on the right hand side of the dependency statement for f or some number statement for τ;
- the types of variables that are quantified over in formulas or set expressions on the right hand side of such a statement;
- the types of the arguments for f or the return types of origin functions for τ.

The symbol graphs for our three examples are shown in Figure 1.6. If the sampling subroutine for a basic RV V hits a basic RV U, then there must be an edge from U's function symbol (or type, if U is a number RV) to V's function symbol (or type) in the symbol graph. This property, along with ideas from [Milch et al., 2005b], allows us to prove the following:

Theorem 1.12
Suppose M is a BLOG model where:

1. uncountable built-in types do not serve as function arguments or as the return types of origin functions;
2. each quantified formula and set expression ranges over a finite set once origin function restrictions are taken into account;
3. the symbol graph is acyclic.

Then M is well-defined. Also, for any evidence instantiation e and query variable Q, the rejection sampling algorithm described in Section 1.6.1 converges to the posterior $P(Q|e)$ defined by the model, taking finite time per sampling step.

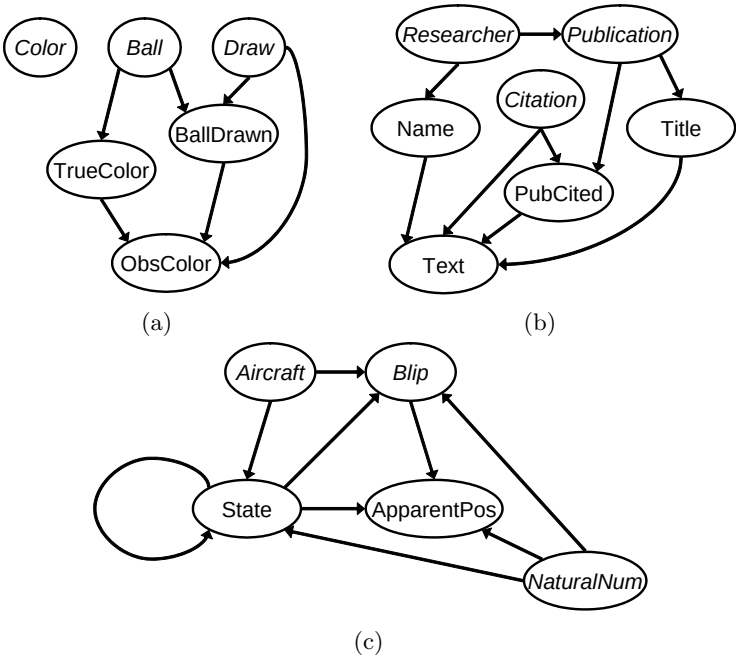


Figure 1.6 Symbol graphs for (a) the urn-and-balls model in Figure 1.1; (b) the bibliographic model in Figure 1.2; (c) the aircraft tracking model in Figure 1.3.

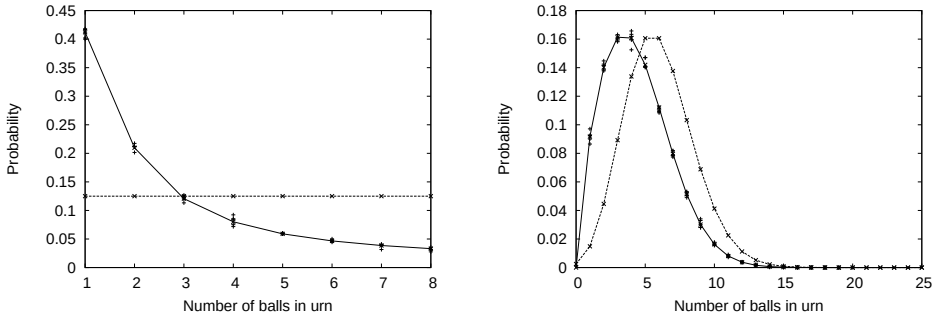
The criteria in Theorem 1.12 are very conservative: in particular, when we construct the symbol graph, we ignore all structure in the dependency statements and just check for the occurrence of function and type symbols. These criteria are satisfied by the models in Figures 1.1 and 1.2. However, the aircraft tracking model in Figure 1.3 does not satisfy the criteria because its symbol graph (Figure 1.6(c)) contains a self-loop from State to State. The criteria do not exploit the fact that $\mathsf{State}(a, t)$ depends only on $\mathsf{State}(a, \mathsf{Pred}(t))$, and the nonrandom function Pred is acyclic. Friedman et al. [1999] have already dealt with this issue in the context of probabilistic relational models; their algorithm can be adapted to obtain a stronger version of Theorem 1.12 that covers the aircraft tracking model.

1.6.3 Experimental results

Milch et al. [2005b] describe a guided likelihood weighting algorithm that uses backward chaining from the query and evidence nodes to avoid sampling irrelevant variables. This algorithm can also be adapted to BLOG models. We applied this algorithm for Example 1.1, asserting that 10 balls were drawn and all appeared blue, and querying the number of balls in the urn. Figure 1.7(a) shows that when the prior for the number of balls is uniform over $\{1, \ldots, 8\}$, the posterior puts more weight on small numbers of balls; this makes sense because the more balls there

are in the urn, the less likely it is that they are all blue. Figure 1.7(b), using a Poisson(6) prior, shows a similar but less pronounced effect.

Note that in Figure 1.7, the posterior probabilities computed by the likelihood weighting algorithm are very close to the exact values (computed by exhaustive enumeration of possible worlds with up to 170 balls). We were able to obtain this level of accuracy using runs of 20,000 samples with the uniform prior, and 100,000 samples using the Poisson prior. On a Linux workstation with a 3.2 GHz Pentium 4 processor, the runs with the uniform prior took about 35 seconds (571 samples/second), and those with the Poisson prior took about 170 seconds (588 samples/second). Such results could not be obtained using any algorithm that constructed a single fixed BN, since the number of potentially relevant TrueColor $[b]$ variables is infinite in the Poisson case.



(a) (b)

Figure 1.7 Distribution for the number of balls in the urn (Example 1.1). Dashed lines are the uniform prior (a) or Poisson prior (b); solid lines are the exact posterior given that 10 balls were drawn and all appeared blue; and plus signs are posterior probabilities computed by 5 independent runs of 20,000 samples (a) or 100,000 samples (b).

1.7 Related Work

Gaifman [1964] was the first to suggest defining a probability distribution over first-order model structures. Halpern [1990] defines a language in which one can make statements about such distributions: for instance, that the probability of the set of worlds that satisfy Flies(Tweety) is 0.8. *Probabilistic logic programming* [Ng and Subrahmanian, 1992] can be seen as an application of this approach to Horn-clause knowledge bases. Such an approach only defines *constraints* on distributions, rather than defining a unique distribution.

Most first-order probabilistic languages (FOPLs) that define unique distributions

fix the set of objects and the interpretations of (non-Boolean) function symbols. Examples include relational Bayesian networks [Jaeger, 2001] and Markov logic models [Domingos and Richardson, 2004]. Prolog-based languages such as probabilistic Horn abduction [Poole, 1993], PRISM [Sato and Kameya, 2001], and Bayesian logic programs [Kersting and De Raedt, 2001] work with *Herbrand models*, where the objects are in one-to-one correspondence with the ground terms of the language (a consequence of the unique names and domain closure assumptions).

There are a few FOPLs that allow explicit *reference uncertainty*, i.e., uncertainty about the interpretations of function symbols. Among these are two languages that use indexed RVs rather than logical notation: BUGS [Gilks et al., 1994] and indexed probability diagrams (IPDs) [Mjolsness, 2004]. Reference uncertainty can also be represented in probabilistic relational models (PRMs) [Koller and Pfeffer, 1998], where a "single-valued complex slot" corresponds to an uncertain unary function. PRMs are unfortunately restricted to unary functions (attributes) and binary predicates (relations). Probabilistic entity-relationship models [Heckerman et al., 2004] lift this restriction, but represent reference uncertainty using relations (such as $\mathsf{Drawn}(d, b)$) and special mutual exclusivity constraints, rather than with functions such as $\mathsf{BallDrawn}(d)$. Multi-entity Bayesian network logic (MEBN) [Laskey and da Costa, 2005] is similar to BLOG in allowing uncertainty about the values of functions with any number of arguments.

The need to handle unknown objects has been appreciated since the early days of FOPL research: Charniak and Goldman's plan recognition networks (PRNs) [Charniak and Goldman, 1993] can contain unbounded numbers of objects representing hypothesized plans. However, external rules are used to decide what objects and variables to include in a PRN. While each possible PRN defines a distribution on its own, Charniak and Goldman do not claim that the various PRNs are all approximations to some single distribution over outcomes.

Some more recent FOPLs do define a single distribution over outcomes with varying objects. IPDs allow uncertainty over the index range for an indexed family of RVs. PRMs and their extensions allow a variety of forms of uncertainty about the number (or existence) of objects satisfying certain relational constraints [Koller and Pfeffer, 1998, Getoor et al., 2001] or belonging to each type [Pasula et al., 2003]. However, there is no unified syntax or semantics for dealing with unknown objects in PRMs. MEBNs take yet another approach: an MEBN model includes a set of unique identifiers, for each of which there is an "identity" RV indicating whether the named object exists.

Our approach to unknown objects in BLOG can be seen as unifying the PRM and MEBN approaches. Number statements neatly generalize the various ways of handling unknown objects in PRMs: number uncertainty [Koller and Pfeffer, 1998] corresponds to a number statement with a single origin function; existence uncertainty [Getoor et al., 2001] can be modeled with two or more origin functions (and a CPD whose support is $\{0, 1\}$); and domain uncertainty [Pasula et al., 2003] corresponds to a number statement with no origin functions. There is also a correspondence between BLOG and MEBN logic: the tuple representations in

a BLOG model can be thought of as unique identifiers in an MEBN model. The difference is that BLOG determines which objects actually exist in a world using number variables rather than individual existence variables.

Finally, it is informative to compare BLOG with the IBAL language [Pfeffer, 2001], in which a program defines a distribution over outputs that can be arbitrary nested data structures. An IBAL program could implement a BLOG-like generative process with the outputs viewed as logical model structures. But the declarative semantics of such a program would be less clear than the corresponding BLOG model.

1.8 Conclusions and Future Work

BLOG is a representation language for probabilistic models with unknown objects. It contributes to the solution of a very general problem in AI: intelligent systems must represent and reason about objects, but those objects may not be known *a priori* and may not be directly and uniquely identified by perceptual processes. Our approach defines generative models in which first-order model structures are created by adding objects and setting function values; everything else follows naturally from this design decision.

Much work remains to be done on BLOG. The inference algorithms presented in this paper are not practical for any but the smallest examples. For real-world problems, we expect to employ Markov chain Monte Carlo (MCMC) techniques (see, e.g, Gilks et al. [1996]), simulating a Markov chain over possible worlds. More precisely, these algorithms must use partial descriptions of possible worlds: in a model with infinitely many RVs, a world cannot be represented explicitly as a full instantiation. We plan to implement a general Gibbs sampling algorithm for BLOG models, using some of the same techniques as the BUGS system [Gilks et al., 1994]. However, for models with unknown objects, we expect to obtain faster convergence with Metropolis-Hastings algorithms [Metropolis et al., 1953] using proposal distributions that split and merge objects [Jain and Neal, 2004]. For now, it appears that these proposal distributions will need to be designed by hand to propose reasonable splits and merges (e.g., merging publications with similar or identical titles), as was done in [Pasula et al., 2003]. However, we have implemented a general Metropolis-Hastings inference engine for BLOG that maintains the state of the Markov chain and computes acceptance probabilities for any given proposal distribution. In the future, we plan to explore adaptive MCMC techniques (see, e.g., [Haario et al., 2001] and references therein).

Another important question is how to design BLOG models that will lead to accurate inferences from real-world data. For the citation matching problem, Pasula et al. [2003] obtained state-of-the-art accuracy using reasonably simple prior distributions for publication titles and author names, estimated from BibTeX files and U.S. Census data (these results are competitive with the discriminative approach of Wellner et al. [2004]). It is not so clear how to estimate the prior distributions for the numbers of objects of various types, such as researchers and publications.

Pasula et al. [2003] simply used a log-normal distribution, which has a very large variance. As an alternative to defining such a prior distribution, one could use the nonparametric version of BLOG proposed by Carbonetto et al. [2005], which incorporates Dirichlet process mixture models.

Finally, perhaps the most interesting questions about BLOG have to do with learning. Parameter estimation, even from partially observed data, is conceptually straightforward: the sampling-based inference algorithms described above can serve as the basis for Monte Carlo expectation-maximization (EM) algorithms [Wei and Tanner, 1990]. But learning the structure of BLOG models is an exciting open problem. In other statistical relational formalisms, techniques have been proposed for discovering dependencies that hold between attributes of related objects [Friedman et al., 1999, Popescul et al., 2003]. We believe that extensions of these techniques can be applied to BLOG. The ultimate goal, however, is to develop algorithms that can hypothesize new attributes, new relations, and even new types of objects. BLOG provides a language in which such hypotheses can be expressed.

References

P. Carbonetto, J. Kisyński, N. de Freitas, and D. Poole. Nonparametric Bayesian logic. In *Proc. 21st Conf. on Uncertainty in AI*, pages 85–93, 2005.

E. Charniak and R. P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *ICML Workshop on Statistical Relational Learning and Its Connections to Other Fields*, 2004.

N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proc. 16th Int'l Joint Conf. on AI*, pages 1300–1307, 1999.

H. Gaifman. Concerning measures in first order calculi. *Israel J. Math.*, 2:1–18, 1964.

L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In *Proc. 18th Int'l Conf. on Machine Learning*, pages 170–177, 2001.

W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Chapman and Hall, London, 1996.

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177, 1994.

H. Haario, E. Saksman, and J. Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.

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

M. Jaeger. Complex probabilistic modeling with recursive relational Bayesian networks. *Annals of Math and AI*, 32:179–220, 2001.

S. Jain and R. M. Neal. A split-merge Markov chain Monte Carlo procedure for the Dirichlet process mixture model. *J. Computational and Graphical Statistics*, 13:158–182, 2004.

K. Kersting and L. De Raedt. Adaptive Bayesian logic programs. In *Proc. 11th Int'l Conf. on Inductive Logic Programming*, 2001.

D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proc. 15th National Conf. on Artificial Intelligence*, pages 580–587, 1998.

K. Lambert. Free logics, philosophical issues in. In E. Craig, editor, *Routledge Encyclopedia of Philosophy*. Routledge, London, 1998.

K. B. Laskey and P. C. G. da Costa. Of starships and Klingons: Bayesian logic for the 23rd century. In *Proc. 21st Conf. on Uncertainty in AI*, pages 346–353, 2005.

N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chemical Physics*, 21:1087–1092, 1953.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. 19th Int'l Joint Conf. on AI*, pages 1352–1359, 2005a.

B. Milch, B. Marthi, D. Sontag, S. Russell, D. L. Ong, and A. Kolobov. Approximate inference for infinite contingent Bayesian networks. In *10th Int'l Workshop on Artificial Intelligence and Statistics*, 2005b.

E. Mjolsness. Labeled graph notations for graphical models. Technical Report 04-03, School of Information and Computer Science, UC Irvine, 2004.

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

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.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Francisco, revised edition, 1988.

A. Pfeffer. IBAL: A probabilistic rational programming language. In *Proc. 17th Int'l Joint Conf. on AI*, pages 733–740, 2001.

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

A. Popescul, L. H. Ungar, S. Lawrence, and D. M. Pennock. Statistical relational learning for document mining. In *Proc. 3rd IEEE Int'l Conf. on Data Mining*, pages 275–282, 2003.

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

R. W. Sittler. An optimal data association problem in surveillance theory. *IEEE Trans. Military Electronics*, MIL-8:125–139, 1964.

G. C. G. Wei and M. A. Tanner. A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. *J. American Stat. Assoc.*, 85:699–704, 1990.

B. Wellner, A. McCallum, F. Peng, and M. Hay. An integrated, conditional model of information extraction and coreference with application to citation matching.

In *Proc. 20th Conf. on Uncertainty in AI*, 2004.