BLOG: Probabilistic Models with Unknown Objects

Brian Milch Bhaskara Marthi Stuart Russell
David Sontag Daniel L. Ong
Andrey Kolobov

Computer Science Division
University of California
Berkeley, CA 94720
{*milch,bhaskara,russell,dsontag,dlong,karaya1*}*@cs.berkeley.edu*

December 13, 2004

Abstract

In many practical problems—from tracking aircraft based on radar data to building a bibliographic database based on citation lists—we want to reason about an unbounded number of unseen objects with unknown relations among them. Probabilistic models for such scenarios are difficult to represent with Bayesian networks (BNs), or even with existing first-order probabilistic languages. This report describes a new language, called BLOG (Bayesian LOGic), for modeling scenarios with unknown objects. A well-defined BLOG model specifies a probability distribution over model structures of a first-order logical language; these model structures can include varying sets of objects. A BLOG model can also be viewed as describing a *contingent Bayesian network* (CBN): a directed graphical model with conditions on the edges indicating the contexts in which they are active. A CBN corresponding to a BLOG model may contain cycles and have infinitely many variables. Nevertheless, we give general conditions under which such a BLOG model defines a unique distribution over model structures. We also present a likelihood weighting algorithm that performs approximate inference in finite time per sampling step on any BLOG model that satisfies these conditions.

1 Introduction

Human beings and artificially intelligent entities must convert streams of sensory input into some understanding of what's out there and what's going on in the world. That is, they must make inferences about the entities and events that underlie their observations. No pre-specified list of objects is given; a central aspect of the task is inferring the existence of objects that were not known initially to exist.

In many existing applications of AI and statistics, this problem of unknown objects is engineered away or resolved in a preprocessing step. However, there are applications of great practical importance where reasoning about unknown objects 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 in the population were known in advance. *Record linkage*, a task undertaken by an industry of more than 300 companies, involves matching entries across multiple databases. The 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 objects that may be generating them.

Probability models for such tasks are not new; for example, generative 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 application 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 data records. It seems natural, therefore, to seek a *formal language* in which to express probability models that allow for unknown objects. In recent years, the creation of formal languages such as graphical models (Pearl, 1988) has accelerated algorithm research and facilitated model sharing and comparison, reusable software, more flexible modeling, rapid application development, and automated model selection (learning). As we argue in Sec. 9, however, there is not yet a formal representation language that can describe probability models with unknown objects in a compact and intuitive way. This report introduces BLOG (Bayesian LOGic), a representation language that meets this requirement.[1]

We explain the basics of BLOG syntax and semantics in Sec. 2 using three examples of increasing complexity: a toy urn-and-balls setup, a record linkage task involving citations and publications, and a multi-target tracking task. Sec. **??** defines the semantics of BLOG more formally. We explain how each BLOG model defines a particular first-order logical language, as well as a set of model structures of this language that serve as the *possible worlds* of the BLOG model. The main technical issue here is choosing a representation for unknown objects so that the set of possible worlds is not too large. Next, we explain how a BLOG model defines a set of constraints on the probability distribution over possible worlds. The BLOG model is *well-defined* if these constraints specify a unique distribution.

In existing work on first-order probabilistic models (Friedman et al., 1999; Kersting & De Raedt, 2001), the standard technique for proving that a model is well-defined is to show that it reduces to a well-defined Bayesian network (BN). A BN is well-defined if it is acyclic and each of its variables has finitely many ancestors. However, many useful BLOG models do not correspond to such well-defined BNs. Thus, in Sec. **??**, we introduce *contingent Bayesian networks* (CBNs): directed graphical models where some edges are labeled with conditions indicating the contexts in which they are active (see (Milch et al., 2005) for a summary of results on CBNs). We then derive certain conditions under which a CBN is *structurally well-defined* and hence guaranteed to

[1] BLOG was introduced briefly and informally in (Milch et al., 2004).

define a unique distribution, even if it contains cycles or infinite ancestor sets. So we can show that a BLOG model is well-defined by showing that it corresponds to some structurally well-defined CBN. Sec. **??** explores the relationship between BLOG models and CBNs in more detail.

In Sec. 6 we discuss how to assert evidence about unknown objects in a BLOG model. Then in Sec. 7, we provide a likelihood weighting algorithm for approximate inference in BLOG models (the algorithm actually operates on a CBN corresponding to the model). Experimental results obtained with this algorithm are presented in Sec. 8. The point of this algorithm is not that it is particularly efficient: indeed, sampling algorithms designed by hand for particular problems take orders of magnitude fewer steps to converge. However, the algorithm applies to any structurally well-defined CBN, and it takes finite time per sampling step even if the CBN has infinitely many nodes.

Our technical development in this paper is limited to discrete random variables (which may have countably many possible values). There is no reason why continuous random variables cannot be used in BLOG: indeed, some of the variables in our multi-target tracking example take real vectors as values. However, extending our results to the continuous case raises a number of technical issues that we have not yet fully resolved, and that would be burdensome to the reader in any case. So a treatment of continuous variables is postponed to a future paper.

2 Example BLOG Models

2.1 Balls in an Urn

We begin with a very simple example from (Russell, 2001) to develop intuition.

Example 1. *Consider an urn containing a random number of balls — say, a number chosen from a Poisson distribution. Each ball is black with probability 0.5; otherwise it is white. We repeat the following experiment M times (where M is an exogeneously determined constant): draw a ball at random from the urn, observe its color, and put it back in the urn. We cannot tell two identically colored balls apart. Also the room is dark, so each time we draw a ball, our color observation is mistaken with probability 0.8. Given a sequence of observed colors, our task is to compute posterior distributions for queries such as, "How many balls are in the urn?" or, "Was the same ball drawn on draws 1 and 2?"*

We stated above that a BLOG model defines a distribution over model structures of a first-order logical language. Specifically, BLOG uses *typed* first-order languages, where the objects are divided into types and each predicate and function symbol takes arguments of particular types. For Ex. 1, we use three types: Color, Ball and Draw. The function symbols are TrueColor, which takes a Ball and returns a Color; BallDrawn, which takes a Draw and returns a Ball; and ObsColor, which takes a Draw and returns a Color. The language for this example also includes constant symbols Black and White to refer to the colors, and constant symbols Draw1, Draw2, ... for the draws. Using this

language, we can assert evidence such as $\mathsf{ObsColor}(\mathsf{Draw1} = \mathsf{Black})$ and ask a query such as $\mathsf{TrueColor}(\mathsf{BallDrawn}(\mathsf{Draw1}))$.

Note that the standard assumptions of *unique names* and *domain closure*, which are made in Prolog-based formalisms such as Bayesian logic programs (Kersting & De Raedt, 2001), do not hold here. The terms $\mathsf{BallDrawn}(\mathsf{Draw1})$ and $\mathsf{BallDrawn}(\mathsf{Draw2})$ may refer to the same ball, and some balls may not be referred to by any term.

A model structure for a typed first-order language maps each type to a set of objects, called its *extension*, and each function symbol to a function, called its *interpretation*. For instance, the interpretation of BallDrawn is a function from the extension of Draw to the extension of Ball. Constant symbols are treated as zero-ary function symbols, so the interpretation of Black is a function that maps the empty tuple to an element of the extension of Color.

To define a probability distribution over model structures, we use a generative process that constructs a model structure step by step. Certain aspects of the model structure are nonrandom and determined before the process starts: in this example, the extension of Color consists of two objects referred to by Black and White, and the extension of Draw consists of one object for each constant symbol of type Draw. Initially there are no objects of type Ball. However, the first step in the generative process is to sample a number from a Poisson distribution, and add that many objects to the extension of Ball. Then, for each ball b, we set the value of TrueColor on b by sampling from a Bernoulli distribution. Next, for each draw d, we sample $\mathsf{BallDrawn}(d)$ uniformly from the extension of Ball, and sample $\mathsf{ObsColor}(d)$ conditioning on $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$. This yields a complete model structure.

With this background, it should be easy to understand the BLOG model for this scenario, shown in Fig. 1. The first 6 lines define the typed first-order language used in this model, including the return types and argument types of the functions. The `guaranteed` statements both introduce constant symbols and specify non-random aspects of the model: the probability distribution will be restricted to model structures where there are two colors and four draws.

The remaining lines of the BLOG model describe the generative process. Line 7 is a *number statement*, specifying a distribution for the number of objects of a certain type. Lines 8–12 are *dependency statements*: a dependency statement specifies how values are chosen for a function on each of its possible tuples of arguments. Like a BN, a BLOG model mainly defines the dependency structure, and allows the conditional probability distributions (CPDs) to be defined elsewhere. In our implementation, CPDs are instances of Java classes: thus, the notation `Poisson[6]` in line 7 instructs the BLOG interpreter to create an instance of the Poisson class with mean 6.

Lines 9 and 12 illustrate passing arguments to CPDs. In the first case the argument is the set {`Ball b`}; the `UniformChoice` CPD defines a uniform distribution over whatever set is passed into it. In the second case, the argument is the term `TrueColor(BallDrawn(d))`, which serves as a parent of $\mathtt{ObsColor}(d)$.

This example also illustrates the use of the special value null. Consider the case where we happen to sample zero as the number of balls in the urn. Then what is $\mathsf{BallDrawn}(\mathsf{Draw1})$? To handle such cases, we allow the interpretations of functions to map their arguments to a special value null; the `UniformChoice` CPD returns a distribution concentrated on null when its argument is an empty set. The use of

```
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 Black, White;
6 guaranteed Draw Draw1, Draw2, Draw3, Draw4;

7 #Ball ~ Poisson[6]();

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

9 BallDrawn(d) ~ UniformChoice[]({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: BLOG model for the urn-and-balls scenario of Ex. 1 with four draws from the urn.

null values also explains why we don't need an "else" case in the last dependency statement in Fig. 1: by convention, function values default to null when none of the "if" conditions in a dependency statement are satisfied.

2.2 Record Linkage

Example 2. *In this task we are given some citations taken from the "works cited" lists of publications in a certain field. We wish to recover the true sets of researchers and publications in this field. For each publication, we want to infer the true title and author list; for each researcher, we want to infer a full name. We also want to determine which publication each citation refers to.*

Matching up co-referring citations is a standard record linkage task; Ex. 2 adds the task of matching authors and inferring their full names. The model we present in this paper is a simplified version of the one used to obtain state-of-the-art results in (Pasula et al., 2003) (those results have subsequently been improved upon by (Wellner et al., 2004)).

The BLOG model in Fig. 2 defines the following generative process for this example. First, generate some number of researchers, and choose a name for each one. Then generate a number of publications, depending on the number of researchers (publications are not generated by individual researchers). For each publication, choose the number of authors, then choose each author by sampling uniformly from the set of researchers that have not been included earlier in the author list. Also, choose the title of

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

random String Name(Researcher);
random Researcher Author(Publication, NaturalNum);
random String Title(Publication);
random Publication PubCited(Citation);
random String NameAsCited(Citation, NaturalNum);
random String TitleAsCited(Citation);
random String CitString(Citation);

nonrandom Boolean Less(NaturalNum, NaturalNum) = LessThan;

guaranteed Citation Citation1, Citation2, Citation3, Citation4;

#Researcher ~ NumResearchersDistrib();

Name(r) ~ NamePrior();

#Publication ~ NumPubsDistrib(#{Researcher r});

NumAuthors(p) ~ NumAuthorsDistrib();

Author(p, i)
    if Less(i, NumAuthors(p)) then
        ~ UniformSample({Researcher r : !EXISTS NaturalNum j
                          (Less(j, i) & Author(p, j) = r)});

Title(p) ~ TitlePrior();

PubCited(c) ~ UniformChoice(Publication p);

NameAsCited(c, i)
    if Less(i, NumAuthors(p)) then
        ~ NameObs(Name(Author(PubCited(c), i)));

TitleAsCited(c) ~ TitleObs(Title(PubCited(c)));

CitString(c)
    ~ CitDistrib({(i, NameAsCited(c, i)) for NaturalNum i :
                   Less(i, NumAuthors(p))},
                  TitleAsCited(c));
```

Figure 2: BLOG model for the bibliographic database construction task of Ex. 2 with four observed citations.

the publication. We choose not to model how the given list of citations is generated; instead, we assume the number of citations is fixed. For each citation, choose its cited publication uniformly at random from the set of publications. Sample the author names and title that appear in the citation according to some string corruption models. Finally, construct the whole citation string, given the corrupted author names and title.

This BLOG model is conceptually similar to the one for urn-and-balls, but illustrates several additional features of the BLOG language. First, it uses the *built-in types* Boolean, String and NaturalNum. These types exist in every BLOG model and always have their obvious extensions. Line 9 introduces a *non-random function* (specifically a non-random *predicate*, which is a function whose return type is Boolean). The interpretations of non-random functions are fixed before the generative process begins. Like CPDs, these non-random interpretations are specified by Java classes—in this case `LessThan`. The function computed by this particular Java class is extremely simple, but it could be more complicated, doing significant computation or looking up values in a large table. For instance, in a genetics problem where one knows the family tree and does not want to define a prior distribution over trees, one could let Mother and Father be non-random functions that look up values in a data file.

Finally, Fig. 2 shows the range of expressions that can be used as arguments to CPDs. The CPD argument in line 13 is a *cardinality expression* #{`Researcher r`}, representing the size of a set. In lines 17–18, we pass into `UniformSample` the set of researchers that satisfy a certain logical formula. Earlier we saw the set {`Ball b`} being used as a CPD argument in Fig. 1; the logical formula was omitted there because it was just the unrestrictive formula `true`. The last kind of CPD argument in this example is a *tuple set*, in lines 26–27. This set consists of the tuples `(i, NameAsCited(c, i))` for those values of `i` that satisfy the given formula. Passing this set of integer-string pairs into the CPD provides more information than just passing in a set of strings, because the CPD gets to know the order of the strings (BLOG currently has no feature for passing a *sequence* into a CPD).

2.3 Multi-Target Tracking

Our final example is an augmented version of the standard multi-target tracking problem.

Example 3. *Consider tracking an unknown number of aircraft over an area containing some unknown number of air bases. At each time step, each aircraft is either flying at some position and velocity toward some destination, or on the ground at some base. If an aircraft is on the ground at time t, it has some probability of taking off, with some other air base as its destination. If an aircraft is in the air near its destination, it may land. We assume each aircraft has a home base where it is located at time 0, and no aircraft enter or leave the area.*

We observe the area with radar: flying aircraft may appear as 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 at a given time step. We do not observe the identity of the aircraft that generated a blip.

```
 1 type Aircraft;  type AirBase;  type RadarBlip;

 2 random R2Vector Location(AirBase);
 3 random AirBase CurBase(Aircraft, NaturalNum);
 4 random R6Vector State(Aircraft, NaturalNum);
 5 random AirBase Dest(Aircraft, NaturalNum);
 6 random Boolean TakesOff(Aircraft, NaturalNum);
 7 random Boolean Lands(Aircraft, NaturalNum);
 8 random R3Vector ApparentPos(RadarBlip);

 9 nonrandom NaturalNum Pred(NaturalNum) = Predecessor;
10 nonrandom NaturalNum Greater(NaturalNum, NaturalNum) = GreaterThan;

11 generating AirBase HomeBase(Aircraft);
12 generating Aircraft BlipSource(RadarBlip);
13 generating NaturalNum BlipTime(RadarBlip);
```

Figure 3: Header of BLOG model for the multi-target tracking task of Ex. 3. The model is split into two figures just because it does not fit on a single page.

If we did not include air bases, this would be a standard multi-target tracking task with detection failure and clutter. Adding air bases makes the problem more realistic in that the aircraft are not just doing a random walk (with momentum) in the sky; they will tend to correct their courses to go toward their destinations. Also, we can query the locations of the air bases, which may be very important to us.

In Examples 1 and 2, all the non-guaranteed objects of each type were added in a single step of the generative process; then the values of functions such as BallDrawn and PubCited were set by sampling from the set of balls or publications. This approach does not make sense for the multi-target tracking example. Instead, we would like to say that at each time step, each aircraft generates some (possibly empty) set of radar blips. One of the most important features of BLOG is that it allows us to represent scenarios where objects generate objects, as illustrated in the BLOG model for Ex. 3 in Figures 3 and 4.

This model describes the following generative process. First generate some air bases, and choose a location for each one. Then, for each air base b, generate some number of aircraft that have b as their home base. Then sample a trajectory for each aircraft a starting at time 0. To start with, set $\mathsf{TakesOff}(a, 0)$ and $\mathsf{Lands}(a, 0)$ to false; $\mathsf{CurBase}(a, 0)$ to $\mathsf{HomeBase}(a)$; and $\mathsf{InFlight}(a, 0)$ to false. Then for each subsequent time step t, sample $\mathsf{TakesOff}(a, t)$, $\mathsf{Lands}(a, t)$, $\mathsf{CurBase}(a, t)$, $\mathsf{InFlight}(a, t)$, $\mathsf{State}(a, t)$ and $\mathsf{Dest}(a, t)$. This sampling must be done in order by time step because, for instance, $\mathsf{State}(a, t)$ depends on $\mathsf{State}(a, \mathsf{Pred}(t))$. Next, for each aircraft a and time t, generate a set of radar blips whose source is a and whose timestamp is t. Also, for each time t, generate a set of "false alarm" blips whose timestamp is t but whose source is null. Finally, for each radar blip r, sample an apparent position (this is the

```
#AirBase ~ NumBasesDistrib();

Location(b) ~ UniformLocation();

#Aircraft: (HomeBase) -> (b)
    ~ NumAircraftDistrib();

TakesOff(a, t)
    if (Greater(t, 0) & (!InFlight(a, Pred(t))) then
        ~ TakeoffBernoulli();

Lands(a, t)
    if (Greater(t, 0) & InFlight(a, Pred(t))) then
        ~ LandingDistrib(State(a, Pred(t)), Location(Dest(a, Pred(t))));

CurBase(a, t)
    if (t = 0) then = HomeBase(a)
    elseif TakesOff(a, t) then = null
    elseif Lands(a, t) then = Dest(a, Pred(t))
    else = CurBase(a, Pred(t));

InFlight(a, t) <-> (CurBase(a, t) = null);

State(a, t)
    if TakesOff(a, t) then
        ~ InitState(Location(CurBase(a, Pred(t))))
    elseif InFlight(a, t) then
        ~ StateTransition(State(a, Pred(t)), Location(Dest(a, t)));

Dest(a, t)
    if TakesOff(a, t) then
        ~ UniformChoice({AirBase a})
    elseif InFlight(a, t) then
        = Dest(a, Pred(t));

#RadarBlip: (BlipSource, BlipTime) -> (a, t)
    if InFlight(a, t) then ~ NumDetectionsDistrib();

#RadarBlip: (BlipTime) -> (t)
    ~ NumFalseAlarmsDistrib();

ApparentPos(r)
    if (BlipSource(r) = null) then ~ FalseDetectionDistrib()
    else ~ ObsDistrib(State(BlipSource(r), BlipTime(r)));
```

Figure 4: Body of BLOG model for the multi-target tracking scenario of Ex. 3.

apparent position of the aircraft that might have generated the blip, based on range and bearing measurements and the position of the radar installation).

To describe generative steps where objects generate objects, this BLOG model uses more complicated number statements than we have seen so far. The number statement for type Aircraft in line 3 of Fig. 4 says that for each air base b, there is a set of aircraft a such that $\mathsf{HomeBase}(a) = b$. We (and the BLOG interpreter) can tell that b refers to an air base because AirBase is the return type of HomeBase. Looking back at Fig. 3, we can see that HomeBase is declared not as an ordinary random function symbol, but as a *generating function* symbol. This is because there are no steps in our generative process where we sample a value for the HomeBase function on an existing aircraft: instead, the value of $\mathsf{HomeBase}(a)$ is set when a is generated.

The number statement for radar blips in line 27 of Fig. 4 describes how radar blips are generated by pairs (a, t) where a is an aircraft and t is a time step. In general, a non-guaranteed object is generated by a tuple of *generating objects* (this tuple may be empty), and is tied back to those generating objects by generating functions. When a number statement does not mention one of the generating functions for the type of object being generated, such as BlipSource in line 30, that function takes the value null on the generated objects.

The other syntactic feature introduced in Fig. 4 is the use of an equals sign to represent a deterministic CPD. This occurs most prominently in the dependency statement for CurBase, lines 11–15. The expression "`= term`" is simply an abbreviation for "$\sim$ `EqualsCPD(term)`", where `EqualsCPD` is a deterministic CPD that takes one argument and returns a distribution that assigns probability 1 to that argument. In a dependency statement for a predicate, such as InFlight (line 16), we use "`<-> formula`" rather than "`= term`".

At this point the reader should have an intuitive understanding of what BLOG models look like and what they mean. The main point is that a BLOG model defines a generative process with two kinds of steps: those that add objects to the world (described by number statements) and those that set the value of a function on some tuple of arguments (described by dependency statements). Many forms of relational uncertainty, including cases where the value of a function is sampled from a set of non-guaranteed objects and cases where objects generate other objects, can be described in a unified syntax.

3 Syntax

This section contains an exhaustive discussion of the BLOG syntax. First, we will present the full BLOG model for our aircraft example. Next, using this example we will give a preview of the semantics of a BLOG model, which will hopefully uncover some intuition behind the construction of programs in this language. Finally, we will delve into the details of syntax itself while constantly referring back to the example program to illustrate their use.

3.1 The Working Example

Throughout the ensuing discussion we will employ the aircraft example to clarify our ideas. For its high-level description please refer to section [Brian's section]. The BLOG code is given in Fig. 1.

3.1.1 The Example

3.1.2 Semantics of the Example

Let us take a glance at the way the presented example describes the aircraft domain. First, it introduces the *types* of the objects in the domain, which are Aircraft, AirBase, and RadarBlip (there are also inherently present types like integers and vectors, but we will not concentrate on them right now). To generate objects of each type, BLOG models use devices called *POPs*. The examples of such devices are #Aircraft, #Airbase, and #Radarblip. POPs describe the distribution from which the number of objects of each type is sampled. Note that the POPs may have "parameters" (as is the case with #Aircraft and #RadarBlip). POPs with parameters give rise to objects that stand in certain relations to the already existing objects, so, for instnace, for some air base b we can generate some number of aircraft whose home base is b.

Consider how the behavior of the radar is modeled by POPs. The Type RadarBlip hastwo POPs. One of them (line 27) probabilistically produces some number of radar blips coming from the actual aircraft. The underlying probability distribution enables us to take account of the fact that not all aircraft may be detected at the given time step. Moreover, the second POP (line 30) introduces more ambiguity by making some false radar blips (in real life, those may be reflections from the clouds).

Producing objects using POPs with parameters is not the only way to state the dependencies between these objects, however. A very convenient way to do this is to use *functors*. In our case, the functors are InFlight, State, Dest, Location, TakesOff, Lands,ApparentPos, and CurBase. They allow us to link up objects of the domain in a very elegant way.

Suppose we are trying to track aircraft a with a radar. We know that at any time t, a can be on the ground, taking off, flying, or landing. The functors CurBase, TakesOff, Lands, and InFlight say what a may be doing at time t, given some information about its state at $t-1$. The functor State probabilistically updates the state of the aircraft at each time step. Combined with the simulation of the radar, the information provided by these functors lets us guess the position of a (which we may know to be the aircraft that took off from airbase b but which is merely a dot on the radar screen indistinguishable from other dots) and perhaps predict its destination.

In the next section, we examine more closely the mechanics of BLOG and how BLOG's language elements describe the generative process for a particular model.

3.2 BLOG Language Elements

We could imagine constructing an outcome in the above domain in the following way:

1. First, we sample an integer denoting the number of air bases from some prior distribution NumBasesDistrib().

2. Next, we sample a location from a prior UniformLocation() for every air base and the number of aircraft that have the base as their homebase.

3. For each aircraft thus generated and for every point in time starting at $t = 0$ we would like to determine whether the aircraft is in flight, is taking off, landing, or is on the ground.

4. For each aircraft in the air we need to specify a transition function that predicts the position of the aircraft at time *t* given its position at *t-1*.

5. Finally, we need to decide whether the given aircraft gets detected by a radar blip at time step *t*.

The result of this generative process is a possible world; the set of all possible worlds can be used to do inference, thereby answering various queries about the domain that the user may wish to ask.

BLOG is aiming to provide a natural way to describe the generative process precisely. It uses the following language elements to achieve this goal:

- *Types*. BLOG is a typed language so every object in a BLOG possible world belongs to one of several types, e.g. Aircraft, AirBase, RadarBlip, etc. Each type in a BLOG model is either built-in or user-defi ned (see section 2.3.1).

- *Guaranteed objects*. Every type may have zero or more guaranteed objects associated with it. The existence of these objects lets the user reflect the knowledge that, for instance, there is some minimum number of objects of the given type in any possible world in the domain under consideration. In the aircraft example, the user may wish to say that there are at least 2 air bases on the monitored area (this fact is stated on line **??** of Fig 1). All objects of built-in types are automatically guaranteed; they need not be listed explicitly (see section 2.3.3).

- *Potential object patterns (POPs)*. Each type may have one or several POPs associated with it. A POP is a generator that determines the number objects of the given type satisfying certain properties in the given world. For example, POP #AirBase( ) → ( ) samples the number of air bases present in some BLOG world, while #Aircraft(Homebase) → (b) proposes, for every air base b generated by #AirBase, the number of Aircraft that have that air base as their home base. Thus, POPs would carry out steps 1 and 3 of our informal plan.

- *Typed functors*. Functors are a class of language elements uniting predicates and functions of fi rst-order logic. In each possible world, every functor has an interpretation, i.e. a value for every tuple of its arguments. The value of the given functor for every argument tuple is determined at some possible-world-generation stage. In the aircraft tracking domain, for instance, after sampling the number of air base objects in step 1, we would use Location functor to sample a location for each of them.

Thus, the BLOG language elements help us naturally carry out our possible-world-generation plan. We now proceed to consider each langauge feature and the syntax for it in detail.

3.3 Syntax

3.3.1 Structure of a BLOG model.

A fully-specified BLOG model contains the description of all dependencies for generating a possible world; model file names have extension ".mblog". For an example of a BLOG model file please refer to Fig.1. A model may be described with up to 6 types of statements:

1. Type declarations
2. Guaranteed object declarations
3. Functor declarations
4. Non-random functor definitions
5. Dependency statements
6. Number statements

All statements are separated by semicolons. By convention, all type declarations appear before statements of all other types. A functor declaration must precede that functor's dependency statement and any other dependency statement that invokes this functor.

3.3.2 Types

Clearly, every BLOG program should have one or more types to define a meaningful model. Every type must be introduced by exactly one type declaration of the form

```
type type_name;
```

BLOG types can be subdivided into built-in and user-defined ones. Built-in types have been precoded in the interpreter. They are:

- Boolean, denotes logical true and false values;
- Real, denotes the set of all real numbers;
- NaturalNum, denotes the set of all integers numbers;
- RkVector ($k \geq 2$), denotes the set of all k-dimensional vectors;
- String, denotes the set of all strings.

The BLOG representation for entities of these types is discussed in Appendix A. Note that although BLOG does support sets, they are not a built-in type; moreover, they should not be defined as a type at all. Defining sets into a type would enable functors (see section 2.3.4) to take sets as arguments, which is not a feature of first-order logic.

3.3.3 Guaranteed Objects

In some cases, the user knows the minimum number of objects of some type in the domain. BLOG allows one, for every type, to name all objects of this type known to exist as follows:

```
guaranteed type_name  = id_1, ... , id_n;
```

In our example, the statement on line **??** of Fig 1 says that there are at least two air bases, AB1 and AB2, on the monitored territory.

3.3.4 Functors

There are three types of functors in BLOG: random, non-random, and generating.

Random functors describe the relations that may change from one world to another. The value of a random functor on a given tuple of arguments is sampled at some stage of the generative process according to the "recipe" outlined in the functor's definition, called "*dependency statement*" in BLOG terminology.

Non-random functors model relations that remain constant over all worlds. Since the number of non-guaranteed objects of every type is allowed to change between worlds (the non-guaranteed objects of a given type are generated by the type's POPs, see section 2.3.8), it follows that non-random functors may only be defined for tuples all of whose elements are guaranteed objects.

Generating functors correspond to functions in first-order logic. They are allowed to take exactly one argument and are never defined explicitly. A more thorough discussion of their role is presented in section 2.3.7.

3.3.5 Random Functors

Random functors (RF) must be both declared and endowed with a dependency statement.

3.3.5.1 Random Functor Declarations.

The declaration of a functor f should precede the dependency statement of any other functor that invokes f in its body. An RF declaration is analogous to a function prototype in C. It specifies the functor type, name, return type, and argument types, e.g.

```
random  ret_type f_name( arg_type_1, ... arg_type_n );
```

Note that the declaration does not contain the parameter identifiers. Functor overloading is not allowed.

Examples of random functor declarations are in lines 2-8 of Fig 1.

3.3.5.2 Dependency Statements.

A dependency statement consists of a header and a body. The header contains the functor name and parameter names. The body is a sequence of if-else statements. The condition of each if-else clause is a formula (see section 2.3.10). The body of an if-else clause contains a reference to a probability function (discussed later in this section). The executed clause is the first one whose condition holds. A dependency statement may look like this:

```
func_name(arg_lst )
    if cond_1 then ~PF_1[param_lst_1](PF_arg_lst_1)

    else if cond_2 then ~PF_2
    ...
    else ~PF_n[param_lst_n](PF_arg_lst_n)
```

where the template *arg_lst* is a list of variable identifiers separated by commas, and *PF_arg_lst*$_i$ is a list of terms.

There a few technicalities to mention:

- Every probability function (PF) *PF* referred to in a BLOG program must be represented as a java class *PF*.java and included in package blog. PFs may take a (possibly empty) list of parameters each of which is of a built-in type, and a (possibly empty) list of arguments. For the restrictions on PF arguments, see section 2.3.11.
- The body of a clause is a PF invokation preceded by '$\sim$'. Sometimes it is convenient to make the body of a clause a term or a formula, in which case it is preceded by '='. The "= arg" is an abbreviation for "$\sim$EqualsPF(arg)," where EqualsPF returns a distribution that assigns probability 1 to its argument (the argument, as already mentioned, can be a formula or a term). For instance, on line 10 of Fig 1 functor Lands invokes the PF "LandingDistrib", while on line 13 one of Curbase's if-else clauses has a functor application term in its body.

If the value of a functor always depends on the same formula, term or probability function invokation, the dependency statement may take the form

```
func_name(arg_lst){
    ~PF[param_lst](PF_arg_lst)
};
```

or

```
func_name(arg_lst){
    =term
};
```

In the aircraft example, functors Location (Fig 1, line 2) and InFlight(Fig 1, line 16) have dependency statements that look as just described.

If the default value of a functor is null, the concluding else statement may be omitted, as in the dependency statements of functors State (Fig 1, line 17) and Dest (Fig 1, line 22).

3.3.6 Nonrandom Functors

Non-random functors are fully specified by their declaration:

```
nonrandom ret_type f_name(arg_type_1,..., arg_type_n) = "class_name";
```

A non-random functor definition should be supplied by the user in a .java file by the name *class_name*. A non-random functor declaration example is on line 9 of Fig. 1.

3.3.7 Generating Functors

As already mentioned, generating functors need only be declared, not defined. The set of generating functors that have type T as their argument type *is generating set for type T*. From now on it will be referred to as S_{GT}; its meaning will become clear in section 2.3.8. The generating functor declaration template is:

```
generating ret_type f_name(arg_type_1, ... , arg_type_n);
```

Examples of generating functor declarations are on lines 11-13 of Fig 1.

3.3.8 Potential Object Patterns (POPs)

The objects of a given user-defined type may be introduced in two ways:

- if the user is certain about the existence of some objects, they may be listed explicitly as guaranteed objects;
- if the number of objects of a given type is unknown, it can be sampled from some prior probability distribution.

POPs can be viewed as generators of objects of the given user-defined type that allow us to introduce objects in the latter case. POPs are declared and defined simultaneously by *number statements*. A number statement header looks like this:

```
#Type_name:(gen_func_lst) → (param_lst);
```

Here, *Type_name* is the name of the type whose objects are generated, *gen_func_lst* is list of the generating functor identifiers separated by commas, and *param_lst* is the list of parameters separated by commas, equal in length to the list of generating functors. Every generating functor must take only one argument. The argument should be of the type *Type_name*. The header is followed by the number statement body enclosed in $\{\}$. The body is a list of if/elseif clauses with the same syntax as that of a dependency statement. The examples of number statements can be found on lines 3, 27, and 30 of Fig 1.

During the generative process, a POP can be used to generate a number of objects of the given type with the unifying property that each of them of is mapped by the POP's i-th generating functor in the i-th object given in the parameter list. Thus, the order of the parameters does matters.

As one may notice in the aircraft example, each type T may have more than one associated POP (RadarBlip has two, lines 27 and 30 of Fig 1). The POPs may differ from each other in the number of generating functors and in the generating functors themselves. If the set of generating functors of a POP is a proper subset of S_{GT} (generator set for type T) then the remaining functors in S_{GT} are assumed to map the objects spawned by this POP to null (i.e.all remaining functors in S_{GT} have an undefined value on these objects). Consider the example in Fig. 1. The first POP for type RadarBlip (line 27) generates, for each aircraft *a* and time step *s*, the number of radar blips (0 or 1) produced by *a* at time *s*. The second POP produces, for each time step, the number of false detections - radar blips that were not caused by any aircraft.

Thus, every object of a user-defined type in a BLOG possible world is either guaranteed or has been generated by a POP, in which case it stands in certain relations to some other objects in this possible world.

3.3.9 Terms

Terms in BLOG correspond to terms in first-order logic. There are several classes of BLOG terms:

- a functor application - follows the template *f_name(*arg_1*, . . . ,* arg_n*)*. In case the functor has an empty argument list, the parentheses may be dropped.
- a variable - an ordinary identifier;
- a built-in constant term - an object of a built-in type, which can be:
  - a Boolean;
  - a Real;

- an NaturalNum;
- a Vector;
- a String.

The syntax for objects of built-in types is discussed in Appendix A.

3.3.10 Formulas

BLOG formulas correspond to sentences in first-order logic and can be subdivided into several categories:

- an atomic formula - any boolean-valued term;
- a conjunction formula - follows the template $formula_1$ & $formula_2$;
- a disjunction formula - follows the template $formula_1$ | $formula_2$;
- a negation formula - follows the template *!formula*;
- an equality formula - follows the template $term_1$ = $term_2$;

To emphasize/impose a particular order of operations, formulas may be enclosed in parentheses.

3.3.11 PF Argument Specifications

A PF can accept more categories of arguments than functors, namely:

- formulas;
- terms;
- set specifications;
- implicit set cardinality specifications.

Terms in formulas have been discussed in sections 2.3.9 and 2.3.10, respectively. Sets can come in 3 various flavours:

- Explicit sets. These are (possibly empty) lists of terms, where the list surrounded by {}:

  {$term_1$, ... , $term_n$};

- Implicit sets. All such set specifications follow the template:

```
type_name var_id: opt_formula};
```

where *var_id* is a variable identifier, *type_name* is that variable's type name, and *opt_formula* is an optional condition (formula) that this variable satisfies. The template above thus describes a set of all objects in the current possible world that satisfy condition *formula*. On line 24 of Fig. 1, PF UniformChoice takes as an argument the implicitly-specified set of all air bases. Note that the condition has been omitted.

- Tuple sets. They look as follows:

```
{(term_1, ... , term_n) for
        type_name_1 var_id_1,..., type_name_m var_id_m: opt_formula};
```

The $term_i$ parametrizes the i-th element if the tuple. The collection of all $term_i$'s relies on variables var_id_i's of corrsponding types $type\_name_i$'s. There variables satisfy the (optional) condition *opt_formula*.

The implicit set cardinality can be passed to the PF in an argument of the form:

```
#implicit_set;
```

where *implicit_set* is the implicit set whose cardinality is being calculated. One should always make sure that the *implicit_set* is finite.

3.3.12 Scope of BLOG variables

The variables in a BLOG program can be introduced in RF/POP headers and in PF argument set specifications. The scope of the variables introduced in the header is the whole dependency/number statement of the corresponding RF/POP. The scope of variables introduced while specifying a set is restricted to that set specification (i.e. between the '{ }' that enclose the specification). Since the scope of the header-introduced variables extends to all set specifications in the corresponding dependency/number statement, the names of the variables introduced for describing a set must be distinct from those of the header-introduced variables.

4 Syntax and Semantics

So far we have introduced the syntax and semantics of BLOG informally. This section clarifies exactly what syntactic constructs make up a BLOG model, and formalizes the semantics that we presented intuitively in Sec. 2. A BLOG model consists of a sequence of *statements*, each terminated by a semicolon (white space is irrelevant). There are five kinds of statements: type declarations, function declarations (of which there are three subtypes, *random*, *nonrandom* and *generating*), guaranteed object statements, number

statements, and dependency statements. Statements of various types can be interleaved in the model file, but a type or function cannot be used in a statement before it is declared.

As stated earlier, a BLOG model M defines a probability distribution over model structures of a particular typed first-order logical language. The typed first-order language $\mathcal{L}_M$ is defined by the type declarations, function declarations, and guaranteed object statements (which implicitly declare constant symbols to refer to the guaranteed objects). To determine the particular set of model structures over which M defines a distribution (that is, the *possible worlds* of M), we also need to look at the left-hand sides of number statements, which tell us what non-guaranteed objects may exist. Finally, the dependency statements and number statements specify constraints on the distribution over possible worlds. We will now discuss each of these aspects of a BLOG model in detail.

4.1 Typed first-order language

BLOG is based on typed (or sorted) first-order logic (see, for example, (Enderton, 2001)). A BLOG model M defines a particular typed first-order language $\mathcal{L}_M$, which plays a role in both syntax and semantics: the terms and formulas in the BLOG model are terms and formulas of $\mathcal{L}_M$, and the BLOG model defines a distribution over model structures of $\mathcal{L}_M$.

4.1.1 Logical language of a BLOG model

A typed first-order language $\mathcal{L}$ is a tuple $(T_\mathcal{L}, F_\mathcal{L}, s_\mathcal{L})$ where $T_\mathcal{L}$ is a set of *type symbols*, $F_\mathcal{L}$ is a set of *functor symbols* (by a "functor" we simply mean a function or a predicate), and $s_\mathcal{L}$ maps each element of $F_\mathcal{L}$ to a *type signature*. The type signature of a k-ary functor f is a tuple $(\tau_1, \ldots, \tau_k, \tau_{k+1})$, where $\tau_1, \ldots, \tau_k$ are the argument types of f and τ_{k+1} is the return type of f. We assume that $T_\mathcal{L}$ always includes a special type Boolean; a functor whose return type is Boolean is called a *predicate*. Constant symbols are treated as zero-ary functor symbols.

A BLOG model M specifies a particular typed first-order language $\mathcal{L}_M$. The identifiers used for type and function symbols can be any strings of alphanumeric characters (including the underscore), as long as they do not begin with a digit.

Definition 1. *For a* BLOG *model M, the typed first-order language $\mathcal{L}_M = (T_{\mathcal{L}_M}, F_{\mathcal{L}_M}, s_{\mathcal{L}_M})$ is defined as follows. $T_{\mathcal{L}_M}$ consists of those symbols τ such that M contains a type declaration*

```
type τ;
```

plus the built-in type symbols Boolean, NaturalNum, Integer, String, Real, *and* RkVector *for $k \geq 2$. For each functor declaration in M of the form:*

```
random τ_{k+1} f(τ_1, ..., τ_k);
nonrandom τ_{k+1} f(τ_1, ..., τ_k) = classname;
```

Built-In Type	Built-In Constant Symbols
`Boolean`	`true`, `false`
`NaturalNum`	numerals `0`, `1`, `2`, ...
`Integer`	numerals ... `-1`, `+0`, `+1`, ...
`String`	strings enclosed in double quotes
`Real`	numerals of the form `4.2`, `1.0`, `3.2e-10`, etc.
Rk`Vector` ($k \geq 2$)	lists of real numbers enclosed in square brackets, separated by commas, e.g., `[3.2, 9.1, 7.3]`

Table 1: Built-in constant symbols of the various built-in types. User-defined symbols are required to be identifiers (strings of alphanumeric characters not beginning with a digit), but the BLOG parser treats these built-in constant symbols specially.

`generating` τ_{k+1} $f(\tau_1, \ldots, \tau_k)$`;`

there is a symbol f in $F_{\mathcal{L}_M}$ such that $s_{\mathcal{L}_M}(f) = (\tau_1, \ldots, \tau_{k+1})$. For each guaranteed object statement in M of the form:

`guaranteed` τ $c_1, \ldots, c_n$`;`

there are symbols $c_1, \ldots, c_n$ in $F_{\mathcal{L}_M}$ such that $s_{\mathcal{L}_M}(c_i) = (\tau)$ for $i = 1, \ldots, n$. Finally, the built-in constant symbols c of each built-in type τ (see Table 1) are elements of $F_{\mathcal{L}_M}$ with $s_{\mathcal{L}_M}(c) = (\tau)$.

Note that $\mathcal{L}_M$ makes no distinction between random, nonrandom, and generating functors: that distinction is a feature of BLOG, not typed first-order logic. The built-in constant symbols include, for example, the numerals of types NaturalNum, Integer and Real and the quoted strings of type String. The built-in types and constant symbols are part of $\mathcal{L}_M$ even if they are not used in M; this is a bit counter-intuitive, but does not cause any problems.

4.1.2 Model structures

A model structure of a first-order language is a like a truth assignment for a propositional language: it specifies a way the world might be. Any given first-order sentence (with no free variables) is either true or false in any given model structure. However, a first-order model structure is considerably more complicated than a truth assignment: it consists of a set of objects and an *interpretation function* that maps each functor symbol to a function (of the appropriate arity) on these objects. For instance, in one model structure for Ex. 3, Pred might be interpreted as a function that maps 1 to 0 and 2 to 1; in another model structure, it might be interpreted as, say, the identity function.

A model structure for a *typed* first-order language must specify not just one set of objects, but an *extension function* that maps each type symbol to a set of objects. For convenience, we generalize model structures one step farther. We allow partial functions: a function may be undefined on certain tuples of arguments. We create this

effect by allowing functions to map some argument tuples to a special object called null, which is not in the extension of any type.

Definition 2. *A model structure ω of a typed first-order language $\mathcal{L}$ consists of an extension function that maps each type $\tau \in T_{\mathcal{L}}$ to a set $[\tau]^{\omega}$, and an interpretation function that maps each functor $f \in F_{\mathcal{L}}$ to a function $[f]^{\omega}$. If $s_{\mathcal{L}}(f) = (\tau_1, \ldots, \tau_{k+1})$, then $[f]^{\omega}$ is a function from $[\tau_1]^{\omega} \times \cdots \times [\tau_k]^{\omega}$ to $[\tau_{k+1}]^{\omega} \cup \{\mathsf{null}\}$.*

We assume that $[\mathsf{Boolean}]^{\omega}$ is always the set $\{\mathsf{true}, \mathsf{false}\}$ (thus true and false are both built-in constant symbols and objects in the extension of Boolean). Note that because the extensions of other types can be arbitrary sets, defining the set of all model structures of $\mathcal{L}_M$ would require talking about the non-existent set of all sets. Thus, defining a probability distribution over *all* model structures of $\mathcal{L}_M$ is a lost cause. In Sec. 4.2, we define a restricted set of model structures of $\mathcal{L}_M$ over which we can define a distribution.

4.1.3 Terms and formulas

We will now walk through the syntax and semantics of terms and formulas in typed first-order logic, introducing some notation that we will use later and also explaining our extensions to handle null values. A *term* may be a logical variable, a constant symbol, a functor applied to some tuple of arguments, or the special symbol null (which thus plays a role in both syntax and semantics). An *assignment* a to a set of logical variables in a model structure ω is a function that maps each of the variables to an object in ω: that is, an element of $\cup_{\tau \in T_{\mathcal{L}}} [\tau]^{\omega}$. We will write $(a; x \rightarrow o)$ for the assignment that is the same as a except that x is mapped to o. Similarly, if a and b are two assignments, we will write $(a; b)$ to denote the assignment that is the same as a except that any variables in the domain of b have the values assigned by b.

Definition 3. *Let t be a term of $\mathcal{L}$, ω be a model structure of $\mathcal{L}$, and a be an assignment that maps each free variable of t to an object in ω. Then the* denotation $[t]_a^{\omega}$ *of t in ω under a is defined as follows:*

- *If $t = x$ for some logical variable x, then $[t]_a^{\omega} = a(x)$.*
- *If $t = f(t_1, \ldots, t_k)$ for some functor f with type signature $(\tau_1, \ldots, \tau_{k+1})$, then:*

$$[t]_a^{\omega} = \begin{cases} [f]^{\omega}\left([t_1]_a^{\omega}, \ldots, [t_k]_a^{\omega}\right) & \text{if } [t_i]_a^{\omega} \in [\tau_i]^{\omega} \text{ for } i = 1, \ldots, k \\ \mathsf{null} & \text{otherwise} \end{cases}$$

- *If $t = \mathsf{null}$, then $[t]_a^{\omega} = \mathsf{null}$.*

Note that if the arguments in a term $f(t_1, \ldots, t_k)$ do not denote objects of the correct types —for instance, if some of the arguments denote null —then $f(t_1, \ldots, t_k)$ denotes null as well. Thus, null values propagate upwards.

An *atomic formula* in a typed first-order language is either a term $f(t_1, \ldots, t_k)$ where the return type of f is Boolean, or an *equality formula* of the form $t_1 = t_2$ where t_1 and t_2 are terms. Formulas can be combined with the propositional operators

`!` (negation), `&` (conjunction), `|` (disjunction), and `->` (implication) —note that we are writing the operators as ASCII characters so we can use them in a BLOG model file. A universally quantified formula has the form `ALL` τ `x` ψ where τ is a type, x is a logical variable, and ψ is a formula; existentially quantified formulas have the form `EXISTS` τ `x` ψ. For instance, Fig. 2 uses the existential formula:

```
EXISTS NaturalNum j (Less(j, i) & Author(p, j) = r)
```

The type over which we're quantifying will usually be clear from the contexts in which the quantified variable occurs: in this example, j must be of type NaturalNum because it serves as an argument to Less and as the second argument of Author. However, there are a few cases where the type might be ambiguous, such as when ψ is an equality formula. So a quantified formula must specify the type that it is quantifying over.

We are now ready to talk about when a model structure satisfies a formula under an assignment.

Definition 4. *Let φ be a formula of $\mathcal{L}$, ω be a model structure of $\mathcal{L}$, and a be an assignment that maps all the free variables of φ to objects in ω. Then ω* satisfies *φ under a, written $\omega \models_a \varphi$, if:*

- *φ is an atomic formula $f(t_1, \ldots, t_k)$, and $[f(t_1, \ldots, t_k)]^\omega_a = \mathsf{true}$;*
- *φ is $t_1 = t_2$ and $[t_1]^\omega_a = [t_2]^\omega_a$;*
- *$\varphi = !\psi$ and it is not the case that $\omega \models_a \psi$;*
- *$\varphi = \psi \,\&\, \chi$ and $\omega \models_a \psi$ and $\omega \models_a \chi$;*
- *$\varphi =$* `ALL` *$\tau\ x\ \psi$ and for each $o \in [\tau]^\omega$, $\omega \models_{(a;x\to o)} \psi$.*

Note that if some of the arguments to an atomic formula denote null, then by Def. 3 the atomic formula denotes null, and thus it is not satisfied. We take the standard shortcut of regarding $\psi \mid \chi$ as an abbreviation for $!(!\psi \,\&\, !\chi)$, ψ `->` χ as an abbreviation for $(!\psi) \mid \chi$, and `EXISTS` $\tau\ x\ \psi$ as an abbreviation for `!ALL` $\tau\ x\ !\psi$.

4.1.4 Set expressions

In BLOG, the arguments to CPDs may be terms and formulas, but they may also be other expressions that are not part of typed first-order logic. These expressions denote finite sets of objects, finite multisets of tuples of objects, and the sizes of such collections. Note that these second-order constructs are not terms, and thus cannot serve as arguments to functors.

BLOG currently supports four kinds of set expressions. The simplest is an *explicit set*, which has the form $\{t_1$`,` $\ldots$`,` $t_n\}$ where $t_1, \ldots, t_n$ are terms. An explicit set just denotes the set of objects denoted by $t_1, \ldots, t_n$. Nearly as simple is the *implicit set*, which has the form $\{\tau$ `x :` $\varphi\}$ and denotes the set of objects x of type τ that satisfy the formula φ. For instance, in Fig. 2 we saw the implicit set:

```
{Researcher r : !EXISTS NaturalNum j
                    (Less(j, i) & Author(p, j) = r)}
```

If φ is just the unrestrictive formula `true`, the implicit set can be written in an abbreviated form as $\{\tau$ `x`$\}$.

We can pass a tuple of sets into a CPD by passing a set into each of several arguments. However, there are situations where we want to preserve some correspondence between elements of the different sets —for instance, in Fig. 2, where we wanted to preserve the correspondence between author numbers and name strings. The third kind of set expression, the *tuple set* allows us to preserve correspondences by passing in a set of tuples. More precisely, a tuple set denotes a *multiset* of tuples, where each tuple is associated with a *multiplicity* that specifies how many times it occurs. A tuple set has the form:

$\{$`(`t_1`, ..., `t_n`) for `τ_1 `x`$_1$`, ..., `τ_m `x`$_m$ `: `$\varphi\}$

where $t_1, \ldots, t_n$ are terms, $\tau_1, \ldots, \tau_m$ are types, $x_1, \ldots, x_m$ are logical variables, and φ is a formula. For instance, the tuple set in Fig. 2 is:

```
{(i, NameAsCited(c, i)) for NaturalNum i : Less(i, NumAuthors(p))}
```

To construct the denotation of a tuple set, we iterate over all assignments of appropriately typed objects to $x_1, \ldots, x_m$ such that φ is satisfied. For each such assignment, we evaluate $(t_1, \ldots, t_n)$, and add this tuple to the multiset.

The last kind of set expression is the *cardinality expression*, which has the form $\#r$ where r is any of the other three kinds of set expressions. The cardinality expression denotes the size of the set denoted by r (if r denotes a multiset, then the size is the sum of the multiplicities of all the tuples). To summarize, we give the following formal definition of the semantics of set expressions:

Definition 5. *Let r be a set expression in a* BLOG *model M, ω be a model structure of $\mathcal{L}_M$, and a be an assignment that maps all the free variables of r to objects in ω. Then the denotation of r in ω under a, written $[r]^\omega_a$, is defined as follows.*

- *If r is an explicit set $\{t_1 \ \ldots \ t_n\}$, then $[r]^\omega_a = \{[t_1]^\omega_a, \ldots, [t_n]^\omega_a\}$.*
- *If r is an implicit set $\{\tau\ x : \varphi\}$, then $[r]^\omega_a$ is equal to $\{o \in [\tau]^\omega \ : \ \omega \models_{(a;x\to o)} \varphi\}$, or* null *if that set is infinite.*
- *If r is a tuple set:*

  $$\{(t_1, \ \ldots, \ t_n) \ \text{for} \ \tau_1 \ x_1, \ \ldots, \ \tau_m \ x_m \ : \ \varphi\}$$

  then $[r]^\omega_a$ is a multiset of tuples. The multiplicity of the tuple $(o_1, \ldots, o_n)$ in this multiset is the number of distinct assignments b mapping $x_1, \ldots, x_m$ into

τ	$G_M(\tau)$
Boolean	{true, false}
NaturalNum	$\mathbb{N}$
Integer	$\mathbb{Z}$
String	all finite strings of bytes
Real	$\mathbb{R}$
RkVector ($k \geq 2$)	$\mathbb{R}^k$

Table 2: Guaranteed objects of each built-in type τ in any BLOG model M.

$[\tau_1]^\omega, \ldots, [\tau_m]^\omega$ *such that* $\omega \models_{(a;b)} \varphi$ *and* $([t_1]^\omega_{(a;b)}, \ldots, [t_n]^\omega_{(a;b)}) = (o_1, \ldots, o_n)$. *If the sum of these multiplicities is infinite, then* $[r]^\omega_a =$ null.

- *if* r *is a cardinality expression* $\#r'$ *where* r' *is an implicit set, then* $[r]^\omega_a = |\,[r']^\omega_a\,|$. *If* $[r']^\omega_a$ *is a multiset, then its size is the sum of the multiplicities of its elements; if it is* null, *then its size is also* null.

4.2 Possible worlds

As noted above, the class of all model structures of $\mathcal{L}_M$ is too large to define a probability distribution over. A BLOG model M defines a particular set Ω_M of possible worlds over which a distribution can be defined. This set of possible worlds is determined by the division of functors into random, nonrandom, and generating functors; the definitions of nonrandom functors; the guaranteed object statements; and the potential object patterns (POPs) on the lefthand sides of number statements.

Definition 6. *In a* BLOG *model* M*:*

- *the set* RF_M *of* random functors *is the set of functor symbols whose declarations in* M *begin with* `random`*;*
- *the set* NRF_M *of* nonrandom functors *is the set of functor symbols whose declarations begin with* `nonrandom` *plus all built-in constant symbols and constant symbols introduced by guaranteed object statements;*
- *the set* $\mathrm{GF}_M(\tau)$ *of* generating functors *for type* τ *is the set of functor symbols whose declarations begin with* `generating` *and which take a single argument of type* τ.

4.2.1 Guaranteed objects

Guaranteed objects are objects that exist in every possible world. They may be built-in objects such as strings and real numbers, domain-specific objects such as the colors in the balls-and-urn example, or objects that are known to exist in a particular scenario, such as the draws in the balls-and-urn example or the known individuals in a pedigree inference problem.

Definition 7. *In a* BLOG *model M, the set $G_M(\tau)$ of* guaranteed objects *of type τ is given by Table 2 if τ is a built-in type; otherwise, it is the set of symbols c_i that occur in some guaranteed object statement of the form:*

```
guaranteed τ c1, ..., cn;
```

A guaranteed object statement that introduces constant symbols $c_1, \ldots, c_n$, says that there are n distinct objects that exist in every world and will be referred to using these symbols. It does not explicitly specify what those objects are. Def. 7 just lets the objects be the symbols themselves. Thus, like the built-in symbols true and false, these constant symbols refer to themselves.

4.2.2 Nonrandom functors

Nonrandom functors are functor symbols that have the same interpretation in all possible worlds. They are only defined on guaranteed objects (they yield null on all other objects). The only built-in nonrandom functors are built-in constant symbols such as numerals and quoted strings. However, the modeler can define nonrandom functors such as Pred and Greater that represent functions or relations on built-in objects. Another use for non-random functors is to model exogeneous variables: variables whose values are known and whose probability distributions we do not wish to model. For instance, if we are trying to infer the genotypes of individuals in a known pedigree, we might let the Mother and Father functors be nonrandom.

A nonrandom functor can be defined by an arbitrary piece of Java code. A nonrandom functor definition specifies the name of a Java class that extends the abstract class `NonRandomFunctor`. This class has a method `getValue` that takes a list of objects (the arguments to the functor) and returns an object (the value of the functor on those arguments). Built-in objects are passed to and from `getValue` as Java objects of the appropriate classes (`Boolean`, `Double`, etc.); guaranteed objects introduced in the model are passed as Java objects of class `EnumeratedObject`. For mathematical functors such as Pred, `getValue` will simply apply some mathematical operation to the arguments to compute the return value. For functors that represent exogeneous variables, `getValue` may use a lookup table loaded from some data file. It is an error for `getValue` to return a Java object that does not represent a guaranteed object of the appropriate type (except that it may return a special Java object `Model.NULL` to represent the BLOG value null).

Definition 8. *In a* BLOG *model M, the* nonrandom interpretation $[f]_M$ *of a nonrandom functor $f \in \mathrm{NRF}_M$ with type signature $(\tau_1, \tau_1, \ldots, \tau_{k+1})$ is a function from $G_M(\tau_1) \times \cdots \times G_M(\tau_k)$ to $G_M(\tau_{k+1}) \cup \{\mathsf{null}\}$ defined as follows:*

- *if f is a built-in constant symbol, then $[f]_M(())$ is the built-in object (truth value, number, string, etc.) obtained by parsing the string representation of f, e.g., 47 is interpreted as 47;*
- *if f is a constant symbol c introduced in a guaranteed object statement, then $[f]_M(()) = c$;*

- *otherwise, f is introduced in a nonrandom functor definition of the form:*

  `nonrandom` τ_{k+1} $f(\tau_1, \ldots, \tau_k)$ `= classname;`

  and $[f]_M(o_1, \ldots, obj_k)$ is the value obtained by calling the `getValue` *method on an instance of classname, with arguments corresponding to $o_1, \ldots, o_k$.*

4.2.3 Potential object patterns

If we think of a BLOG model as defining a generative process, then non-guaranteed objects are generated by potential object patterns (POPs). A POP is a tuple $(\tau, f_1, \ldots, f_k)$ where τ is the type of object generated, and $f_1, \ldots, f_k$ are generating functors for τ. For instance, one of the POPs in the aircraft example is $(\mathsf{RadarBlip}, \mathsf{BlipSource}, \mathsf{BlipTime})$. We will sometimes refer to a POP with k generating functors as a k-ary POP; POPs can be zero-ary, like $(\mathsf{AirBase})$ in the aircraft example. The POPs of a BLOG model are specified by the lefthand sides of number statements.

Definition 9. *In a* BLOG *model M, the set $\mathrm{POP}_M(\tau)$ of POPs for type τ is the set of tuples $(\tau, f_1, \ldots, f_k)$ such that M contains a number statement of the form:*

`#`τ `:` $(f_1, \ldots, f_k)$ `->` $(x_1, \ldots, x_k)$ $\ldots$

It is an error for a BLOG model to include two number statements that yield the same POP, or POPs whose lists of generating functors are just permutations of each other. In the generative process, a POP can be applied to a (possibly empty) tuple $(o_1, \ldots, o_k)$ of generating objects, yielding some new objects $q_1, \ldots, q_n$. Each of these objects q is tied back to the generating objects by the generating functors: $f_i(q) = o_i$. For instance, if a radar blip q is generated by applying the POP $(\mathsf{RadarBlip}, \mathsf{BlipSource}, \mathsf{BlipTime})$ to a pair (o_a, o_t) (where o_a is an aircraft and o_t is an integer representing a time step), then $\mathsf{BlipSource}(q) = o_a$ and $\mathsf{BlipTime}(q) = o_t$. If a POP does not include all the generating functors for a type, then the remaining generating functors have the value null on the generated objects. For example, if q is generated by the POP $(\mathsf{RadarBlip}, \mathsf{BlipTime})$, then $\mathsf{BlipSource}(q) = \mathsf{null}$.

We will use a nested tuple $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ to represent a *POP application* where the POP $(\tau, f_1, \ldots, f_k)$ is applied to the tuple $(o_1, \ldots, o_k)$. Given a model structure, we can determine whether any given object was generated by any given POP application, as follows.

Definition 10. *Let M be a* BLOG *model, $(\tau, f_1, \ldots, f_k)$ be a POP in M, $\tau_1, \ldots, \tau_k$ be the return types of $f_1, \ldots, f_k$, and ω be a model structure of $\mathcal{L}_M$. A non-guaranteed object $q \in ([\tau]^\omega \setminus G_M(\tau))$ is* generated by *the POP application $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ in ω if:*

- *$o_i \in [\tau_i]^\omega$ and $[f_i]^\omega(q) = o_i$ for $i \in \{1, \ldots, k\}$; and*
- *for all $g \in (\mathrm{GF}_M(\tau) \setminus \{f_1, \ldots, f_k\})$, $[g]^\omega(q) = \mathsf{null}$.*

Note that if q is generated by $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ in ω, then $\{f_1, \ldots, f_k\}$ is the set of generating functors for τ that yield non-null values when applied to q in ω. So q cannot be generated by an application of any other POP in ω. And because a functor can only map q to a single value in a given model structure, q cannot be generated by the application of this POP to any other tuple of objects in ω. Thus we have the following proposition:

Proposition 1. *In any model structure ω of $\mathcal{L}_M$, any given object q is generated by at most one POP application.*

Prop. 1 expresses a certain uniqueness property, but it does not imply the usual unique names assumption. Several different functors applied to several different tuples of objects may all yield the value q in ω, even though q is only generated by one POP application.

4.2.4 Tuple representations for potential objects

In the model structures that constitute the possible worlds of M, we would like to ensure that each non-guaranteed object is generated by a POP application. More specifically, suppose we look at the objects that generated a given non-guaranteed object, then look at the objects that generated them, and so on. We should eventually reach guaranteed objects or zero-ary POPs —rather than ending up in a cycle or an infinite receding chain of non-guaranteed objects.

This well-foundedness property can be formalized with some careful mathematics. However, to define the set Ω_M of possible worlds, just specifying such properties is not enough. We must actually specify what the potential objects are: that is, what the elements of $[\tau]^\omega$ may be for a type τ and a world $\omega \in \Omega_M$. A convenient way to kill these two birds with one stone is to let the potential objects be nested tuples that encode how they are generated.

Suppose a POP application $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ generates N objects $q_1, \ldots, q_N$. We will require that the generating objects be tuples constructed by adding a number to the end of the POP application: thus $q_n = (\tau, (f_1, o_1), \ldots, (f_k, o_k), n)$. As a concrete example, let's start with the POP $(\mathsf{AirBase})$ in Ex. 3. In each possible world, the objects it generates are $(\mathsf{AirBase}, 1), (\mathsf{AirBase}, 2), \ldots, (\mathsf{AirBase}, N)$ for some N. Applying the POP $(\mathsf{Aircraft}, \mathsf{HomeBase})$ to $(\mathsf{AirBase}, 2)$ generates objects:

$$
\begin{array}{l}
(\mathsf{Aircraft}, (\mathsf{HomeBase}, (\mathsf{AirBase}, 2)), 1) \\
(\mathsf{Aircraft}, (\mathsf{HomeBase}, (\mathsf{AirBase}, 2)), 2) \\
\vdots
\end{array}
$$

Finally, if the second aircraft shown above generates a radar blip at time 8, that blip will be:

$$(\mathsf{RadarBlip}, (\mathsf{BlipSource}, (\mathsf{Aircraft}, (\mathsf{HomeBase}, (\mathsf{AirBase}, 2)), 2)), (\mathsf{BlipTime}, 8), 1)$$

These tuple representations get large and complicated, but they are entirely internal to BLOG; modelers and users never need to deal with them. In fact, users typically

will not know the tuple representations of the objects they observe. For instance, when a user sees a radar blip, he does not know by what aircraft it was generated or even whether it was generated by an aircraft at all. Users refer to objects using only terms of $\mathcal{L}_M$ and Skolem constants, which are introduced in Sec. 6.

Note that the tuple representations of objects are nested to different depths. Let's say that $(\mathsf{AirBase}, 1)$ has depth 1 because it contains a single POP application. The tuple representations of aircraft have depth 2 because they contain nested air base tuples; the radar blip tuple has depth 3 because it contains a nested aircraft tuple of depth 2. Let's also say that guaranteed objects have depth 0. Then for any type τ and natural number d, we can define the set $U_M^d(\tau)$ of objects of type τ that have depths $\leq d$. The set of all potential objects of type τ (guaranteed and non-guaranteed), denoted $U_M(\tau)$, is then the infinite union of these sets.

Definition 11. *In a* BLOG *model M, the sets $U_M^d(\tau)$ of potential objects of type τ with depth $\leq d$ are defined inductively as follows. As the base case, for each type $\tau \in T_{\mathcal{L}_M}$:*

$$U_M^0(\tau) = G_M(\tau)$$

Now for the inductive case, let p be any POP $(\tau, f_1, \ldots, f_k)$, and let $\tau_1, \ldots, \tau_k$ be the return types of $f_1, \ldots, f_k$. We begin by defining the set of objects generated by applications of p at depth $d+1$:

$$U_{M,p}^{d+1}(\tau) = \bigcup_{o_1 \in U_M^d(\tau_1)} \cdots \bigcup_{o_k \in U_M^d(\tau_k)} \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\}$$

Then to complete the inductive case:

$$U_M^{d+1}(\tau) = U_M^d(\tau) \cup \bigcup_{p \in \mathrm{POP}_M(\tau)} U_{M,p}^{d+1}(\tau)$$

Finally, the full set of potential objects *of type τ in M is:*

$$U_M(\tau) \triangleq \bigcup_{d=0}^{\infty} U_M^d(\tau)$$

4.2.5 Set of possible worlds

We are finally ready to define the set Ω_M of possible worlds of a BLOG model M. This definition requires that the extension of each type include the guaranteed objects, but be a subset of the potential objects of that type. If a non-guaranteed object exists in a world, then the objects that generate it must exist as well. Also, the objects generated by a given POP application in a world must be numbered consecutively from 1. The interpretations of nonrandom functors must match their definitions, and the interpretations of generating functors must yield the right values when applied to non-guaranteed objects.

Definition 12. *For a* BLOG *model M, the set Ω_M of* possible worlds *consists of those model structures ω of $\mathcal{L}_M$ such that:*

1. *for each type* $\tau \in T_{\mathcal{L}_M}$*:*
   (a) $G_M(\tau) \subseteq [\tau]^\omega \subseteq U_M(\tau)$*;*
   (b) *for each non-guaranteed object* $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \in [\tau]^\omega$*:*
      i. $o_i \in [\tau_i]^\omega$ *for* $i \in \{1, \ldots, k\}$*, where* τ_i *is the return type of* f_i*;*
      ii. *if* $n > 1$ *then* $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n-1) \in [\tau]^\omega$*;*
2. *for each nonrandom functor* $f \in \mathrm{NRF}_M$*,* $[f]^\omega$ *agrees with* $[f]_M$ *on all tuples that consist only of guaranteed objects, and yields* null *on all other tuples;*
3. *for each type* τ*, generating functor* $f \in \mathrm{GF}_M(\tau)$*, and object* $q \in [\tau]^\omega$*:*
   (a) *if* q *is a non-guaranteed object* $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n)$ *and* $f = f_i$ *for some* $i \in \{1, \ldots, k\}$*, then* $[f]^\omega(q) = o_i$*;*
   (b) *otherwise,* $[f]^\omega(q) = \mathsf{null}$.

Under this definition, the generating functors turn out to be nonrandom: a generating functor applied to an object q yields the same value (a generating object or null) in every possible world where q exists. Also, each POP application may generate a countably infinite set of objects in a given world.

It is intuitively clear that the objects generated by a POP application in a possible world ω (in the sense of Def. 10) should be exactly those objects in ω whose tuple representations have the appropriate form. The following lemma formalizes this intuition and confirms that we have set up our definitions correctly.

Lemma 1. *For each POP* $(\tau, f_1, \ldots, f_k) \in \mathrm{POP}_M(\tau)$*, tuple of objects* $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$ *(where* τ_i *is the return type of* f_i*), and world* $\omega \in \Omega_M$*:*

$$\{q : q \text{ is generated by } (\tau, (f_1, o_1), \ldots, (f_k, o_k)) \text{ in } \omega\} = [\tau]^\omega \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) : n \geq 1, n \in \mathbb{N}\}$$

Proof. Let's consider the conditions under which an object q is generated by the POP application $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ in ω. Part 3 of Def. 12 ensures that an object $q \in [\tau]^\omega$ is generated by this POP application if and only if q is a non-guaranteed object of the form $(\tau, (g_1, q_1), \ldots, (g_\ell, q_\ell), n)$, $\{g_1, \ldots, g_\ell\} \cap \mathrm{GF}_M(\tau) = \{f_1, \ldots, f_k\}$, and $q_i = o_i$ for those values of i such that $g_i = f_i$. Clearly these conditions are satisfied by a tuple of the form $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n)$. So the set on the left is a superset of the set on the right.

Now consider any object q that is generated by $(\tau, (f_1, o_1), \ldots, (f_k, o_k))$ in ω. By Def. 10, q must be a non-guaranteed object in $[\tau]^\omega$. Def. 12 ensures $[\tau]^\omega \subseteq U_M(\tau)$. And by Def. 11, $U_M(\tau)$ consists only of guaranteed objects and tuples of the form $(\tau, (g_1, q_1), \ldots, (g_\ell, q_\ell), n)$ where $(\tau, g_1, \ldots, g_\ell) \in \mathrm{POP}_M(\tau)$, $n \in \mathbb{N}$, and $n \geq 1$. So $q \in [\tau]^\omega$ must be a tuple of this form. Now by the definition of a POP, $\{g_1, \ldots, g_\ell\}$ cannot contain anything other than generating functors for τ. So by the conditions noted above, $\{g_1, \ldots, g_\ell\} = \{f_1, \ldots, f_k\}$. But a model cannot contain two POPs that differ only in the order of their generating functors, so in fact $(g_1, \ldots, g_\ell) = (f_1, \ldots, f_k)$. Then by the last condition above, $q_i = o_i$ for $i \in \{1, \ldots, k\}$. So in fact q is an element of the set on the right, and the proof is complete. □

We will also find a use for the following lemma, which essentially says that the objects generated by a given POP are numbered consecutively starting at 1.

Lemma 2. *For each POP* $(\tau, f_1, \ldots, f_k) \in \mathrm{POP}_M(\tau)$*, tuple of objects* $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$ *(where* τ_i *is the return type of* f_i*), and world* $\omega \in \Omega_M$*:*

$$[\tau]^\omega \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) : n \geq 1, n \in \mathbb{N}\} \\ = \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) : 1 \leq n \leq N, n \in \mathbb{N}\}$$

where $N \in \mathbb{N} \cup \{\infty\}$ *is the cardinality of the set on the lefthand side of the equals sign.*

Proof. Based on Def. 12 part 1(b)ii, a simple induction shows that if $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \in [\tau]^\omega$ then $(\tau, (f_1, o_1), \ldots, (f_k, o_k), m) \in [\tau]^\omega$ for all $1 \leq m \leq n$. So if there were some $n \in \{1, \ldots, N\}$ such that $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \notin [\tau]^\omega$, then no higher-numbered tuples would be in $[\tau]^\omega$ either, and the cardinality would be less than N. Conversely, if there were some $n \geq 1$ not in $\{1, \ldots, N\}$ such that $(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \in [\tau]^\omega$, then the cardinality would be greater than N. □

4.2.6 Basic random variables

Now that we have defined a set of possible outcomes, we can define some random variables (RVs) on this set. In particular, we will define a set of *basic random variables* whose values are sampled in the generative process defined by the BLOG model.

Definition 13. *The set of* basic random variables *of a* BLOG *model* M*, denoted* $\mathbf{V}_M$*, consists of:*

- *For each random functor symbol* $f \in \mathrm{RF}_M$ *with type signature* $(\tau_1, \ldots, \tau_{k+1})$*, and each tuple* $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$*, a* functor application variable $V_f^{(o_1, \ldots, o_k)} : \Omega_M \to (U_M(\tau_{k+1}) \cup \{\mathsf{null}\})$ *defined by:*

$$V_f^{(o_1, \ldots, o_k)}(\omega) = \begin{cases} [f]^\omega(o_1, \ldots, o_k) & \text{if } o_i \in [\tau_i]^\omega \text{ for } i \in \{1, \ldots, k\} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

- *For each type* $\tau \in T_{\mathcal{L}_M}$*, POP* $p = (\tau, (f_1, \ldots, f_k)) \in \mathrm{POP}_M(\tau)$*, and tuple* $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$ *(where* τ_i *is the return type of* f_i*), a* number variable $N_p^{(o_1, \ldots, o_k)} : \Omega_M \to (\mathbb{N} \cup \{\infty\})$ *defined by:*

$$N_p^{(o_1, \ldots, o_k)}(\omega) = |\{q : q \text{ is generated by } (\tau, (f_1, o_1), \ldots, (f_k, o_k)) \text{ in } \omega\}|$$

Note that we do not need anything special in the definition of a number variable $N_p^{(o_1, \ldots, o_k)}$ for the case where the objects $o_1, \ldots, o_k$ don't all exist in ω: by Def. 12 part 1(b)i, the number of generated objects must be zero in that case.

We will write $\mathrm{dom}(X)$ to denote the range of an RV X. An *instantiation* of a set $\mathbf{X}$ of RVs is a function that maps each RV $X \in \mathbf{X}$ to an element of $\mathrm{dom}(X)$. The set of variables to which σ assigns a value will be denoted $\mathrm{vars}(\sigma)$. We will write $\mathrm{dom}(\mathbf{X})$ for the set of all instantiations of a set of RVs $\mathbf{X}$.

Lemma 3. *The function $\omega \mapsto \mathbf{V}_M(\omega)$ on Ω_M is one-to-one.*

Proof. Consider any instantiation σ of $\mathbf{V}_M$, and suppose $\mathbf{V}_M(\omega) = \mathbf{V}_M(\chi)$ for some $\omega, \chi \in \Omega_M$. We must show that $\omega = \chi$. First we will show that for each type $\tau \in T_{\mathcal{L}_M}$, $[\tau]^\omega = [\tau]^\chi$. Consider any POP $p = (\tau, f_1, \ldots, f_k)$ and tuple of objects $(o_1, \ldots, o_k) \in U_M(\tau_1) \times \cdots \times U_M(\tau_k)$ (where τ_i is the return type of f_i). By hypothesis, $N_p^{(o_1,\ldots,o_k)}(\omega) = N_p^{(o_1,\ldots,o_k)}(\chi)$. So:

$$|\{q \,:\, q \text{ is generated by } (\tau, (f_1, o_1), \ldots, (f_k, o_k)) \text{ in } \omega\}| \\ = |\{q \,:\, q \text{ is generated by } (\tau, (f_1, o_1), \ldots, (f_k, o_k)) \text{ in } \chi\}|$$

Thus by Lemma 1,

$$|\,[\tau]^\omega \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\}| \\ = |\,[\tau]^\chi \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\}|$$

Let N be the common cardinality of these two sets. Then Lemma 2 implies that they are both equal to $\{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, 1 \leq n \leq N, n \in \mathbb{N}\}$, so in fact we have:

$$[\tau]^\omega \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\} \\ = [\tau]^\chi \cap \{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\}$$

Now assume for contradiction that some object q is in one of ω, χ but not the other; say $q \in [\tau]^\omega$ but $q \notin [\tau]^\chi$. q cannot be a guaranteed object because $G_M(\tau) \subseteq [\tau]^\chi$. Every element of $[\tau]^\omega$ is in $U_M(\tau)$, so by Def. 11, q must be an element of some set of the form $\{(\tau, (f_1, o_1), \ldots, (f_k, o_k), n) \,:\, n \geq 1, n \in \mathbb{N}\}$. But this contradicts our earlier conclusion that the intersections of $[\tau]^\omega$ and $[\tau]^\chi$ with any set of that form are equal.

So all types have the same extensions in ω and χ. Now consider any functor symbol f. If f is a nonrandom or generating functor, then Def. 12 ensures f has the same interpretation in any two worlds that agree on the extensions of all types. If f is a random functor with type signature $(\tau_1, \ldots, \tau_k)$, then consider any tuple of objects $(o_1, \ldots, o_k) \in [\tau_1]^\omega \times \cdots \times [\tau_k]^\omega$. Since $V_f^{(o_1,\ldots,o_k)}(\omega) = V_f^{(o_1,\ldots,o_k)}(\chi)$, we know $[f]^\omega(o_1, \ldots, o_k) = [f]^\chi(o_1, \ldots, o_k)$. So the worlds agree on the interpretations of all functor symbols. □

Although the mapping from possible worlds to instantiations of the basic variables is one-to-one, it is not onto. There are some instantiations of the basic variables that do not correspond to any possible world: for example, an instantiation might have $N_p^{()} = 2$ where $p = (\mathsf{AirBase})$, but also have $V_f^{(o)} \neq \mathsf{null}$ where $o = (\mathsf{AirBase}, 7)$. This instantiation does not correspond to any world because if $N_p^{()}(\omega) = 2$, then $(\mathsf{AirBase}, 7)$ must not exist in ω, and thus $V_f^{(o)}(\omega)$ must be null.

Definition 14. *An instantiation σ of a set $\mathbf{X}$ of basic RVs is* achievable *in M if there is some $\omega \in \Omega_M$ such that $\mathbf{X}(\omega) = \sigma$. The* achievable values *of a basic RV X given an instantiation σ of $\mathbf{Y}$ are:*

$$\mathrm{range}(X \mid \sigma) \triangleq \{x \in \mathrm{range}(X) \,:\, \exists \omega \in \Omega_M \,(\mathbf{Y}(\omega) = \sigma \text{ and } X(\omega) = x)$$

A basic RV X is determined *by σ if* $|\mathrm{range}(X \mid \sigma)| = 1$.

Thus, if a BLOG model defines a probability measure over the achievable instantiations of $\mathbf{V}_M$, it defines a probability measure over Ω_M.

4.3 Constraints on the probability distribution

In Sec. 3, we discussed the idea of generating a possible world by progressively adding objects and assigning values to functors on tuples of arguments. We can now understand this process as progressively assigning values to basic random variables. The conditional probability distributions (CPDs) used in sampling the variables are specified by the dependency statements and number statements of the BLOG model.

4.3.1 Probability functions

A dependency or number statement specifies CPDs for all functor application variables or number variables that match the lefthand side of the statement; these variables are called the *child variables* of the statement. The righthand side of the statement consists of a sequence of *clauses* $c_1, \ldots, c_n$. Each clause c_i consists of a condition φ_i, a Java class name $classname_i$ representing an *elementary probability function* (EPF), and a tuple of *argument specifications* $(r_{i,1}, \ldots, r_{i,m})$. The condition φ_i may be an arbitrary formula of $\mathcal{L}_M$. The argument specifications r_i may be formulas or terms of $\mathcal{L}_M$ or set expressions of the kind defined in Sec. 4.1.4.

The EPF Java class $classname_i$ must implement the `CondDistrib` interface. This interface has two methods: `isDiscrete` and `getProb`. The arguments to `getProb` are a list of Java objects representing the denotations of argument specifications, and a Java object representing a value x for a child variable. Given these arguments, `getProb` returns a number representing the probability that the child variable takes on the value x. This number is to be interpreted as an actual probability if `isDiscrete` returns true; otherwise it is a density with respect to Lebesgue measure evaluated at x (this is only permitted if the child is a functor application random variable involving a functor whose return type is Real or RkVector). Note that even within a given dependency statement, some of the elementary probability functions may be discrete and others may be continuous.

As with the Java classes that define nonrandom functors (Sec. 4.2.2), built-in BLOG objects are passed to EPFs as objects of standard Java classes such as `Integer`. Guaranteed objects enumerated by the modeler are passed as objects of class `EnumeratedObject`; sets and multisets are also represented by corresponding Java classes. However, unlike nonrandom functor definitions, elementary probability functions can also take non-guaranteed objects as arguments. Non-guaranteed objects are passed in as featureless unique identifiers. The `getProb` method is not allowed to peek at the tuple representations of these non-guaranteed objects; all it can do is check to see whether two of them are equal. More precisely, EPFs must be invariant under permutations of the non-guaranteed objects. This means that any dependencies on how an object was generated must be made explicit in the argument specifications, for instance by having one the arguments be the term $\mathsf{BlipSource}(r)$.

When the child variable may take non-guaranteed objects as values, it is also important that an EPF assign positive probability only to child values x that necessarily exist given the arguments passed in. Since an EPF cannot look at an object's tuple representation, the only way to enforce this requirement is to say that among non-guaranteed objects, `getProb` can assign positive probability only to those that were passed in as part of some EPF argument. For instance, if the only argument is a set expressions $\{\mathsf{Aircraft}\ a\ :\ \mathsf{Dest}(a) = \mathsf{A1}\}$, then the EPF can assign positive probability only to aircraft in the denotation of this set expression.

In a given model structure and under given assignment to logical variables, a sequence of clauses denotes a probability measure for the child variable. This denotation is computed by finding the first clause whose condition is satisfied, evaluating its argument specifications, and then passing the resulting objects to the appropriate `getProb` function. More formally:

Definition 15. *Let M be a* BLOG *model, $c_1, \ldots, c_n$ be the righthand side of a dependency or number statement in M, ω be a model structure of $\mathcal{L}_M$, and a be an assignment mapping the free variables of $c_1, \ldots, c_n$ to objects in ω. Let i be the index of the first clause such that $\omega \models_a \varphi_i$. If such a clause exists, then the denotation $[c_1, \ldots, c_n]_a^\omega$ is the probability measure defined by passing $([r_{i,1}]_a^\omega, \ldots, [r_{i,m}]_a^\omega)$ to $classname_i$.*`getProb`*. Otherwise, $[c_1, \ldots, c_n]_a^\omega$ is the measure that assigns probability 1 to* null *(or to 0 if the child is a number variable, or to* false *if the child is a Boolean functor application variable).*

Given this understanding of what the righthand side of a dependency statement means, we can now define the *probability function* for each basic variable in M.

Definition 16. *Let M be a* BLOG *model and X be a basic RV of M. The* probability function *for X in M, denoted ρ_M^X, is a function that maps each possible world $\omega \in \Omega_M$ to a probability measure on* $\mathrm{range}(X)$*. Either X is a functor application variable $V_f^{(o_1,\ldots,o_k)}$ and M contains a dependency statement:*

$$f(x_1,\ \ldots,\ x_k)\ :\ \ c_1, \ldots,\ \ c_n\ \texttt{;}$$

or X is a number variable $N_p^{(o_1,\ldots,o_k)}$ for some POP $p = (\tau, (f_1, \ldots, f_k))$ and M contains a number statement:

$$\#\tau\ \ :\ \ (f_1,\ \ \ldots,\ \ f_k)\ \texttt{->}\ (x_1, \ldots,\ \ x_k)\ :\ \ c_1, \ldots,\ \ c_n\ \texttt{;}$$

In either case, if all of $o_1, \ldots, o_k$ exist in ω, then $\rho_M^X(\omega) = [c_1, \ldots, c_n]_a^\omega$ where the assignment a maps each x_i to o_i. Otherwise, $\rho_M^X(\omega)$ assigns probability 1 to null *(or 0 if X is a number variable).*

The intention is that ρ_M^X should be the conditional probability distribution (CPD) for X in the joint probability measure defined by M. However, we will reserve the term CPD to refer to a conditional distribution defined by a given joint measure, and just use "probability function" to refer to the functions specified by dependency and number statements.

4.3.2 Support

We have defined the probability function ρ_M^X as a function on possible worlds. But typically, ρ_M^X will yield the same measure on many different possible worlds. Referring back to Ex. 3, let o_r be a radar blip $(\mathsf{RadarBlip}, (\mathsf{BlipSource}, o_a), (\mathsf{BlipTime}, o_t))$, Y be the variable $V_{\mathsf{State}}^{(o_a,o_t)}$, and X be the variable $V_{\mathsf{ApparentPos}}^{(o_r)}$. Then any two worlds that agree on Y also agree on ρ_M^X. In such cases, we say that any instantiation of Y *supports* X.

Definition 17. *An instantiation σ of* $\mathbf{Y}$ supports *a basic RV X in a* BLOG *model M if for all $\omega, \chi \in \Omega_M$:*

$$((\mathbf{Y}(\omega) = \sigma) \wedge (\mathbf{Y}(\chi) = \sigma)) \rightarrow (\rho_M^X(\omega) = \rho_M^X(\chi))$$

If σ is achievable and supports X, we will write $\rho_M(X \mid \sigma)$ to denote the probability measure $\rho_M^X\omega$ on $\mathrm{range}(X \mid \sigma)$ shared by all worlds ω in the preimage of σ. The support relation is crucial for our sampling process: we cannot sample X until we have sampled an instantiation that supports X. Thus, the support relation determines the possible orders in which we can sample the basic RVs. In some BLOG models, it will turn out that there is no way to sample the basic RVs —for instance, it may be that the only instantiations supporting X are those that include Y, and at the same time the only instantiations supporting Y are those that include X.

4.3.3 Split trees

In a Bayesian network, it is possible to sample all the variables top-down if the network has a topological ordering: a numbering $X_1, X_2, \ldots$ of the nodes such that any instantiation of $\{X_1, \ldots, X_n\}$ supports X_{n+1}. In a BLOG model, the order in which variables can be sampled may depend on the values chosen for earlier variables. Thus, instead of a single ordering of the variables, we use a *split tree*.

The nodes of a split tree are instantiations of subsets of $\mathbf{V}_M$. The root node is the empty instantiation, and each non-leaf node σ splits on a basic variable X_σ that is supported by σ. The children of σ are instantiations of the form $(\sigma; X_\sigma = x)$ for $x \in \mathrm{range}(X_\sigma \mid \sigma)$. Because $\mathrm{range}(X_\sigma)$ may be infinite, a node may have infinitely many children; also, because the number of basic variables may be infinite, a split tree may have infinitely long paths.

Definition 18. *A tree $\mathcal{T} = (\Sigma, \rightarrow)$ is a* split tree *for M if:*

- *Σ is a set of instantiations of subsets of $\mathbf{V}_M$ that includes the empty instantiation $\varnothing$;*
- *for every $\sigma \in \Sigma$ there is a unique path $\varnothing \rightarrow \cdots \rightarrow \sigma$ in $\mathcal{T}$, and this path is finite;*
- *for every $\sigma \in \Sigma$, either $\{\tau \in \Sigma : \sigma \rightarrow \tau\}$ is empty, or there is some $X_\sigma \in \mathbf{V}_M$ such that σ supports X_σ in M and:*

$$\{\tau \in \Sigma : \sigma \rightarrow \tau\} = \{(\sigma; X_\sigma = x) : x \in \mathrm{range}(X_\sigma \mid \sigma)\}$$

```
1 function SampleFullInst(M, Σ, →)
2     let σ = ∅
3     while {τ : σ → τ} ≠ ∅:
4         sample x from ρ_M(X_σ | ()σ)
5         let σ = (σ; X_σ = x)
```

Figure 5: Algorithm for sampling an instantiation of $\mathbf{V}_M$ given a split tree $\mathcal{T} = (\Sigma, \rightarrow)$ for M.

[TODO: add figure showing part of split tree for balls-and-urn example with a single draw]

Lemma 4. *If $\mathcal{T} = (\Sigma, \rightarrow)$ is a split tree for M, then each instantiation $\sigma \in \Sigma$ is achievable in M.*

Proof. Consider any $\tau \in \Sigma$. If $\tau = \varnothing$, then τ is achievable because every BLOG model has at least one possible world. Otherwise, there is a path $\varnothing \rightarrow \cdots \rightarrow \sigma \rightarrow \tau$ in $\mathcal{T}$. So $\tau = (\sigma; X_\sigma = x)$ for some $x \in \mathrm{range}(X_\sigma \mid \sigma)$. So by the definition of $\mathrm{range}(X_\sigma \mid \sigma)$, τ is achievable. □

A path in $\mathcal{T}$ is any finite or infinite sequence of instantiations $\sigma_1, \sigma_2, \ldots$ such that $\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots$. Intuitively, each path represents a possible run of a sampling algorithm. A path is *truncated* if its last element is not a leaf node in $\mathcal{T}$.

Definition 19. *A split tree $\mathcal{T}$ for M* covers *an RV $X \in \mathbf{V}_M$ if every non-truncated path starting at $\varnothing$ in $\mathcal{T}$ includes an instantiation σ such that either $X \in \mathrm{vars}(\sigma)$ or σ determines X in M. $\mathcal{T}$ covers a set of RVs $\mathbf{X} \subseteq \mathbf{V}_M$ if it covers every $X \in \mathbf{X}$.*

Thus $\mathcal{T}$ covers X if, no matter what path we take in the tree, we eventually split on X. The exception is if X is determined by some instantiation in the path (in the sense of Def. 14; then we don't need to split on X. Note that in a given path, X must be instantiated or determined after a finite number of steps, since every instantiation in a split tree can be reached by a finite path from the root. However, since a split tree can contain infinitely many paths, there may not be any number N such that X is instantiated or determined in every path after N steps.

4.3.4 Sampling a possible world

The notion of a split tree allows us to give our first definition of the semantics of a BLOG model. See Fig. 5.

[What should we actually say here? In general, and specifically for the aircraft example, this sampling algorithm does not terminate. It will sample any given finite set of variables in finite time, but then it's hard to describe it as sampling a possible world. It does terminate for the balls-and-urn and citations example, which actually have split trees where all paths are finite.]

The following lemma guarantees that value x sampled in line 4 is in $\mathrm{range}(X_\sigma \mid \sigma)$, and thus that the new instantiation created in line 5 is a child of the old instantiation in $\mathcal{T}$.

Lemma 5. *If σ is an achievable, finite, self-supporting instantiation that supports X but does not include X, then $\rho_M(X \mid \sigma)$ is concentrated on $\mathrm{range}(X \mid \sigma)$.*

4.4 Declarative semantics

Could the distribution we obtain through our sampling process depend on which split tree we use? Can we give a less procedural characterization of the probability measure defined by a BLOG model?

Definition: BLOG model is *well-defined* if there is a unique probability measure that satisfies these independence assumptions and has the specified probability functions as its CPDs.

Theorem: If a BLOG model has a split tree that covers all the basic random variables, then it is well-defined.

Proof: Cite new version of AI/Stats paper (in progress).

5 Well-Defined BLOG Models

The split tree for a BLOG model is typically infinite. Can we check that a BLOG model has a split tree just by running algorithms on the model itself?

[Lots more to do here; the stuff below is copied from the AI/Stats representation draft. In order to prove these results, we'll have to introduce the whole formalism of contingent BNs. Ordinary BNs don't suffice because for many variables, the set of parents is only context-specifically finite. Ordinary BNs can't handle infinite parent sets.]

One way to ensure that a BLOG model has a split tree is to check that it has an acyclic *symbol graph*.

Definition 20. *The* symbol graph *of a* BLOG *model M is a graph whose nodes are the POPs and random functor symbols of M, and which includes an edge $\beta \rightarrow \alpha$ if:*

- *β (or the type it generates, if it is a POP) appears on the righthand side of the dependency statement for α; or*
- *α is a functor with an argument of the type generated by β; or*
- *α is a POP and one of its generating functors returns the type generated by β.*

Note that we can construct this graph just by syntactic inspection of the dependency statements. When we speak of a type appearing in a dependency statement, we mean as part of a universal or existential quantifier (which must specify which type is being quantified over) or a set specification passed to a CPD. The symbol graph for Ex.3 is shown in Fig. 6.

Proposition 2. *If the symbol graph for M is acyclic, then M is well-defined.*

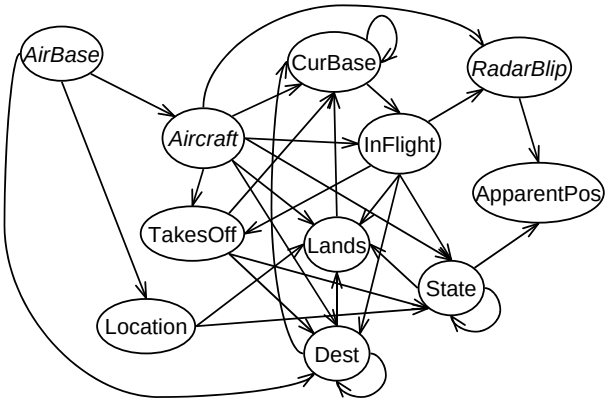

Figure 6: Symbol graph for Example 3.

[Can't prove this proposition as it stands —need to add stipulation that parent sets be context-specifically finite in each possible world.]

Prop. 2 is enough to guarantee that Ex. 2 is structurally admissible, but it cannot make any guarantees about Ex. 3. Cycles arise in the symbol graph for Ex. 3 because some variables depend on the values of variables at the previous time step. But if we expanded the BLOG model into an infinite BN, the edges between these variables would not form cycles; they would form chains. The reason is that there is a partial ordering on time steps, and the nonrandom "-1" function maps its argument to a value that is strictly smaller in this partial ordering.

Friedman et al. (Friedman et al., 1999) have developed an algorithm that takes advantage of such partial orderings to check acyclicity of PRMs; we now present an extension of this algorithm for BLOG. First, we say that a nonrandom unary function f is *reducing* with respect to a partial order $\prec$ if $[f]_M(o) \prec o$ for every appropriately typed object o. Following (Friedman et al., 1999), we assume the modeler has labeled certain nonrandom functions, such as "-1", as reducing. Now, consider for example the cycle $\mathsf{InFlight} \rightarrow \mathsf{Lands} \rightarrow \mathsf{CurBase} \rightarrow \mathsf{InFlight}$ in Fig. 6. Referring to the dependency statements in Fig. **??**, we see that a $\mathsf{Lands}(a_L, t_L)$ variable can only depend on an $\mathsf{InFlight}(a_{IF}, t_{IF})$ variable when $t_{IF} = t_L - 1$. Since "-1" is reducing, it follows that $t_{IF} \prec t_L$. We write this condition on the edge $\mathsf{InFlight} \rightarrow \mathsf{Lands}$. Then for the dependency of $\mathsf{CurBase}(a_{CB}, t_{CB})$ on $\mathsf{Lands}(a_L, t_L)$, we find that it only holds when $t_L = t_{CB}$. We write this condition on the edge $\mathsf{Lands} \rightarrow \mathsf{CurBase}$. Similarly, we write $t_{CB} = t_{IF}$ on the edge $\mathsf{CurBase} \rightarrow \mathsf{InFlight}$. So the conditions on this cycle are $t_{IF} \prec t_L$, $t_L = t_{CB}$, $t_{CB} = t_{IF}$. By transitivity, we get $t_{IF} \prec t_{IF}$, which is impossible; so this cycle in the symbol graph must not correspond to any cycle in the CBN.

Applying this process to each edge in the symbol graph and each argument of the parent functor (or POP), we create a *$\prec$-labeled symbol graph*.

Proposition 3. *If $\prec$ is a well-founded partial order*[2]*, and in the $\prec$-labeled symbol*

[2] A partial order in which every set has a minimal element.

graph for M every cycle is marked with conditions that form a contradiction, then M is well-defined.

Proposition 3 ensures that all three of our example BLOG models define unique probability measures over possible worlds.

6 Evidence

[Section copied from AI/Stats representation draft.]

6.1 Conditioning on Sentences

A BLOG model M for a language $\mathcal{L}$ defines a prior distribution over possible worlds. Thus, it is mathematically straightforward to condition on any sentence φ of $\mathcal{L}$ that encodes some observed evidence. For instance, in Ex. 1, we can assert the sentence $\mathsf{ObsColor}(\mathsf{Draw3}) = \mathsf{Black}$. In Ex. 2, we can assert $\mathsf{Text}(\mathsf{Cit5}) =$ "Casella and Berger. Statistical Inference, 1990" (assuming that text strings are treated as nonrandom constants). Conditioning on a sentence φ just means conditioning on the event $\{\omega \in \Omega_M : \omega \models \varphi\}$.

In our implementation, if φ is simply a predicate applied to a tuple of nonrandom constant symbols, or an equality sentence saying that some functor applied to a tuple of nonrandom constants equals another nonrandom constant, then φ can be asserted directly as an observed value for a node in the CBN. Otherwise, a deterministic node must be added to the CBN, with a CPD that represents φ and an observed value of true.

6.2 Existential Observations and Explicit Observation Models

In scenarios with unknown objects, we often want to assert evidence about objects for which we don't have constant symbols. For instance, in Ex. 3, our language does not include any symbols for radar blips. However, we can assert that there are exactly two blips on the radar screen at time 8 using the sentence:

$$\begin{aligned}\exists \mathsf{RadarBlip}\, b_1\, \exists \mathsf{RadarBlip}\, b_2\, (&\mathsf{BlipTime}(b_1) = 8 \\ &\wedge \mathsf{BlipTime}(b_2) = 8 \\ &\wedge b_1 \neq b_2 \\ &\wedge \forall \mathsf{RadarBlip}\, b'\, (b' = b_1 \vee b' = b_2 \vee \mathsf{BlipTime}(b') \neq 8)) \quad (1)\end{aligned}$$

Conditioning on an existentially quantified sentence is not always the right way to represent one's observations. For example:

Example 4. *Suppose you wander into an unfamiliar wine shop. You are not sure whether this wine shop is fancy or not: a fancy wine shop sells mostly expensive bottles of wine (mean price $40); a cheap wine shop has a few expensive bottles, but sells mostly cheaper ones (mean price $15). In scenario A, you ask the shopkeeper if he has anything over $40, and he says yes. In scenario B, you pull a bottle at random from the shelf and find that it's over $40.*

Obviously the two scenarios are different. The evidence in scenario A is appropriately modeled with a sentence such as:

$$\exists \mathsf{BottleOfWine}\, x\, (\mathsf{In}(x, \mathsf{ThisShop}) \wedge \mathsf{Price}(x) > 40)$$

But note that this observation does not have much effect on one's posterior belief that this is a fancy wine shop; even a cheap wine shop is likely to have something over $40. On the other hand, scenario B is best represented explicitly in the BLOG model (which we do not have space to show in full), using a random zero-ary functor BottleChosen and a dependency statement such as:

$$\mathsf{BottleChosen} : \sim \mathrm{UniformSample}(\{\mathsf{BottleOfWine}\, x \,:\, \mathsf{In}(x, \mathsf{ThisShop})\})$$

One can then condition on the sentence $\mathsf{Price}(\mathsf{BottleChosen}) > 40$. Picking a bottle at random and finding that it is over $40 has a much greater effect on one's belief that one is in a fancy wine shop.

6.3 Skolem Constants

The existentially quantified sentence used to assert evidence about radar blips in the previous section is quite cumbersome. In order to say something more about blip b_2 — for instance, that $\mathsf{ApparentPos}(b_2) = (9.6, 1.2, 32.8)$ —we need to reassert the whole sentence, because we cannot use the variable b_2 outside the existential quantifier. More importantly, we cannot ask queries such as $\mathsf{Dest}(\mathsf{BlipSource}(b_2), 8)$.

In logical reasoning systems, this problem is remedied by the use of Skolem constants. A Skolem constant is a new constant symbol added to the language to replace an existentially quantified variable. If we introduce Skolem constants B1 and B2 for the observed radar blips, we can assert the evidence:

$$\begin{aligned}&\mathsf{BlipTime}(\mathsf{B1}) = 8 \wedge \mathsf{BlipTime}(\mathsf{B2}) = 8 \wedge \mathsf{B1} \neq \mathsf{B2}\\ &\quad \wedge \forall \mathsf{RadarBlip}\, b' \,(b' = \mathsf{B1} \vee b' = \mathsf{B2} \vee \mathsf{BlipTime}(b') \neq 8)\\ &\mathsf{ApparentPos}(\mathsf{B1}) = (10.1, 3.6, 19.5)\\ &\mathsf{ApparentPos}(\mathsf{B2}) = (9.6, 1.2, 32.8)\end{aligned}$$

and then make the query $\mathsf{Dest}(\mathsf{BlipSource}(\mathsf{B2}), 8)$.

But what exactly does it mean to introduce a Skolem constant in a BLOG model? We have extended our first-order language to include a new constant symbol, so we must now define a probability measure over possible worlds that include interpretations for this symbol. What should be the distribution for this symbol's value? One might be tempted to suggest uniform sampling from the set of objects of the appropriate type. But as Ex. 4 illustrates, conditioning on an existentially quantified sentence is very different from conditioning on a result of random sampling. In particular, those two forms of evidence lead to different posterior beliefs about other variables, such as $\mathsf{Fancy}(\mathsf{ThisShop})$.

It turns out that if we view all the evidence assertions where C occurs as a single large sentence $\varphi(\mathsf{C})$, then the correct dependency statement for C is:

$$\mathsf{C} \sim \mathrm{UniformSample}(\{x \,:\, \varphi(x)\}) \tag{2}$$

By convention, UniformSample returns a distribution concentrated on null if the set passed in is empty. So conditioning on the evidence $\mathsf{C} \neq \mathsf{null}$ is the same as conditioning on $\exists x\, \varphi(x)$.

We can now state a probabilistic analogue of Skolem's theorem (Skolem, 1928). First, some notation: if we start with a BLOG model M for a language $\mathcal{L}$, then adding a Skolem constant yields a new language $\mathcal{L}'$. The set of possible worlds Ω'_M for $\mathcal{L}'$ consists of those model structures for $\mathcal{L}'$ that can be obtained by extending an element of Ω_M. Note that each element ω' of Ω'_M extends a unique element of Ω_M, which we call $\text{Proj}_{\mathcal{L}}(\omega')$.

Theorem 1. *Suppose we take a* BLOG *model M and condition on a sentence $\exists x\, \varphi(x)$. If we then extend M to a new model M' by incorporating (2) and conditioning on $\mathsf{C} \neq \mathsf{null}$, we get the same distribution over Ω_M, in the sense that for any measurable event $E \subseteq \Omega_M$:*

$$\mu_{M'}(\{\omega' \in \Omega_{M'} \,:\, \mathit{Proj}_{\mathcal{L}}(\omega') \in E\} \mid \mathsf{C} \neq \mathsf{null}) = \mu_M(E \mid \exists x \varphi(x))$$

Even with Skolem constants, sentence (1) is still unwieldy. There are many cases, such as when we look at a radar screen, when we want to assert that we have observed a set of distinct objects and these are all the objects that have a certain property. Thus, BLOG provides a convenient syntax for such statements. Sentence (1) can be rewritten as:

$$\{\mathsf{RadarBlip}\, r \,:\, \mathsf{BlipTime}(r) = 8\} = \{\mathsf{B1}, \mathsf{B2}\}$$

This syntax makes it less painful to assert evidence and make queries about unknown objects.

7 Inference

In this section we discuss an approximate inference algorithm for CBNs. To get information about a given CBN $\mathcal{B}$, our algorithm will use a few "black box" oracle functions. The function GET-ACTIVE-PARENT(X, σ) returns a variable that is an active parent of X given σ but is not already included in $\text{vars}(\sigma)$. It does this by traversing the decision tree $\mathcal{T}_X^{\mathcal{B}}$, taking the branch associated with σ_U when the tree splits on a variable $U \in \text{vars}(\sigma)$, until it reaches a split on a variable not included in $\text{vars}(\sigma)$. If there is no such variable —which means that σ supports X —then it returns null. We also need the function COND-PROB(X, x, σ), which returns $p_{\mathcal{B}}(X = x|\sigma)$ whenever σ supports X, and the function SAMPLE-VALUE(X, σ), which randomly samples a value according to $p_{\mathcal{B}}(X|\sigma)$.

Our inference algorithm is a form of likelihood weighting. Recall that the likelihood weighting algorithm for BNs samples all non-evidence variables in topological order, then weights each sample by the conditional probability of the observed evidence (Russell & Norvig, 2003). Of course, we cannot sample all the variables in an infi nite CBN. But even in a BN, it is not necessary to sample *all* the variables: the relevant variables can be found by following edges backwards from the query and evidence

variables. We extend this notion to CBNs by only following edges that are active given the instantiation sampled so far. At each point in the algorithm (Fig. 7), we maintain an instantiation σ and a stack of variables that need to be sampled. If the variable X on the top of the stack is supported by σ, we pop X off the stack and sample it. Otherwise, we find a variable V that is an active parent of X given σ, and push V onto the stack. If the CBN is structurally admissible, this process terminates in finite time: condition (A1) ensures that we never push the same variable onto the stack twice, and conditions (A2) and (A3) ensure that the number of distinct variables pushed onto the stack is finite.

As an example, consider the balls-and-urn CBN (Fig. 1). If we want to query N given some color observations, the algorithm begins by pushing N onto the stack. Since N (which has no parents) is supported by $\varnothing$, it is immediately removed from the stack and sampled. Next, the first evidence variable $\mathsf{ObsColor}_1$ is pushed onto the stack. The active edge into $\mathsf{ObsColor}_1$ from $\mathsf{BallDrawn}_1$ is traversed, and $\mathsf{BallDrawn}_1$ is sampled immediately because it is supported by σ (which now includes N). The edge from $\mathsf{TrueColor}_n$ (for n equal to the sampled value of $\mathsf{BallDrawn}_1$) to $\mathsf{ObsColor}_1$ is now active, and so $\mathsf{TrueColor}_n$ is sampled as well. Now $\mathsf{ObsColor}_1$ is finally supported by σ, so it is removed from the stack and instantiated to its observed value. This process is repeated for all the observations. The resulting sample will get a high weight if the sampled true colors for the balls match the observed colors.

Intuitively, this algorithm is the same as likelihood weighting, in that we sample the variables in some topological order. The difference is that we sample only those variables that are needed to support the query and evidence variables, and we do not bother sampling any of the other variables in the CBN. Since the weight for a sample only depends on the conditional probabilities of the evidence variables, sampling additional variables would have no effect.

Theorem 2. *Given a structurally well-defined CBN $\mathcal{B}$, a finite evidence instantiation* **e***, a finite set* **Q** *of query variables, and a number of samples N, the algorithm* CBN-LIKELIHOOD-WEIGHTING *in Fig. 7 returns an estimate of the posterior distribution $P(\mathbf{Q}|\mathbf{e})$ that converges with probability 1 to the correct posterior as $N \to \infty$. Furthermore, each sampling step takes a finite amount of time.*

Proof. Let $\mathcal{X}$ be the set of *minimally* self-supporting instantiations that extend $(\tau; Q = q)$ for any assignment q to Q, such that there exists a path from every variable in the instantiation to $\text{vars}(\tau) \cup Q$. By minimally self-supporting, we mean that removing any variable from an instantiation makes it not self-supporting. Any variable pushed on the stack will eventually be instantiated. Since line 7 only pushes variables that are active (given a variable already on the stack) on the stack, and line 7 ensures that all variables added to σ are supported, σ will be minimally self-supporting.

Let N be the total number of samples. Assume for the moment that in each sampling step we sample all of the non-evidence variables, and let $N(\omega)$ be the count of the number of times we have seen a sample with $\mathcal{V} = \omega$. We will show that the algorithm only needs aggregate counts for the query variables; it does not actually have to sample any variable outside of the minimally self-supporting instantiations. Since we

```
function CBN-LIKELIHOOD-WEIGHTING(Q, e, B, N)
  returns an estimate of P(Q|e)
  inputs: Q, the set of query variables
          e, evidence specified as an instantiation
          B, a contingent Bayesian network
          N, the number of samples to be generated

  W ← a map from dom(Q) to real numbers, with values
    lazily initialized to zero when accessed
  for j = 1 to N do
      σ, w ← CBN-WEIGHTED-SAMPLE(Q, e, B)
      W[q] ← W[q] + w where q = σ_Q
  return NORMALIZE(W[Q])
```

```
function CBN-WEIGHTED-SAMPLE(Q, e, B)
  returns an instantiation and a weight

  σ ← ∅; stack ← an empty stack; w ← 1
  loop
    if stack is empty
      if some X in (Q ∪ vars(e)) is not in vars(σ)
        PUSH(X, stack)
      else
        return σ, w

    while X on top of stack is not supported by σ
      V ← GET-ACTIVE-PARENT(X, σ)
      push V on stack

    X ← POP(stack)
    if X in vars(e)
      x ← e_X
      w ← w × COND-PROB(X, x, σ)
    else
      x ← SAMPLE-VALUE(X, σ)
    σ ← (σ, X = x)
```

Figure 7: Likelihood weighting algorithm for CBNs.

are doing forward sampling from the CPDs,

$$\lim_{N\to\infty} \frac{N(\omega)}{N} \to \prod_{v \in \mathcal{V} \setminus \mathsf{vars}(\tau)} p(v \mid \mathsf{parents}(v)) = S(\omega \backslash \tau).$$

Also, let $weight(\sigma)$ be the weight, calculated by the algorithm, for a given sample σ.

$$weight(\sigma) = \prod_{v \in \mathsf{vars}(\tau)} p(v \mid \mathsf{parents}(v)).$$

The algorithm's estimate of the posterior is:

$$\begin{aligned}
\hat{p}(Q = q \mid \tau) &= \alpha \sum_{\sigma \in \mathcal{X}} weight(\sigma) \sum_{\text{Full inst. } \omega \text{ of} \sigma} N(\omega) \\
&\approx \alpha' \sum_{\sigma \in \mathcal{X}} weight(\sigma) \sum_{\text{Full inst. } \omega \text{ of} \sigma} S(\omega \backslash \tau) \\
&= \alpha' \sum_{\sigma \in \mathcal{X}} weight(\sigma) S_2(\sigma) \\
&= \alpha' \sum_{\sigma \in \mathcal{X}} p(\sigma) \\
&= \alpha' p(Q = q, \tau) \\
&= p(Q = q \mid \tau). \\
S_2(\sigma) &= \prod_{v \in \mathsf{vars}(\sigma) \setminus \mathsf{vars}(\tau)} p(v \mid \mathsf{parents}(v)).
\end{aligned}$$

We have thus shown that, in the limit of the number of samples, this algorithm's estimate of the posterior converges to the true posterior. The algorithm's estimate of the posterior, shown on the first line, only uses counts that are aggregated over the minimally self-supporting instantiations. [TODO: Motivate lines $4 \to 5$. Should be proved, earlier, as a separate theorem.]

To show termination in finite time, first note that no variable is pushed on the stack twice. Since every variable on the stack is a child of the variable above it, the stack always corresponds to some path through $\mathcal{G}\sigma$. By the first structural admissibility condition (A1), $\mathcal{G}\sigma$ is guaranteed to be acyclic. Thus, the stack will never contain two copies of the same variable. A variable is only removed from the stack when it is instantiated in σ. Since line **??** checks to see if a variable about to be added has already been instantiated, and `GET-ACTIVE-PARENT` only returns uninstantiated variables, we can conclude that no variable is pushed on the stack after having previously popped from it. In particular, line **??** will only be called finitely many times.

To show that line 7 is called only finitely many times, we will show that the algorithm traverses a finite forest once. Let each variable pushed on the stack correspond to a vertex in $\mathcal{T}$. When, in lines 7–7, a variable W is pushed on the stack, add the edge (W, V) to $\mathcal{T}$. Since W is an active parent of V, and the third admissibility condition (A3) ensures that every variable has only finitely many active parents, the vertex corresponding to V in $\mathcal{T}$ will have finitely many parents. Furthermore, since every path

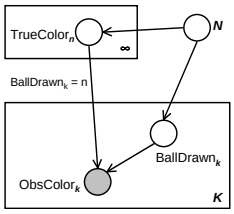

Figure 8: A BN representing the balls and urn model, there are k draws from an urn which contains N balls, where i indexes over k and n indexes over N.

in $\mathcal{T}$ corresponds to some path in $\mathcal{G}\sigma$, A1 ensures that $\mathcal{T}$ will be acyclic. Thus, $\mathcal{T}$ is a forest with at most $|E \cup Q|$ finitely branching trees. By König's Lemma, each tree in $\mathcal{T}$ is infinite iff it has an infinite path. Since an infinite path in $\mathcal{T}$ would violate the second admissibility condition (A2), we conclude that $\mathcal{T}$ is finite. □

8 Experiments

We ran two sets of experiments using the likelihood weighting algorithm of Fig. 7. Both use the balls and urn setup from Ex. 1 and Fig. 8. The first experiment estimates the number of balls in the urn given the colors observed on 10 draws; the second experiment is an identity uncertainty problem. In both cases, we run experiments with both a noiseless sensor model, where the observed colors of balls always match their true colors, and a noisy sensor model, where with probability 0.2 the wrong color is reported.

The purpose of these experiments is to show that inference over an infinite number of variables can be done using a general algorithm in finite time. We show convergence of our results to the correct values, which were computed by enumerating equivalence classes of outcomes with up to 100 balls (see (**?**) for details). More efficient sampling algorithms for these problems have been designed by hand (Pasula, 2003); however, our algorithm is general-purpose, so it needs no modification to be applied to a different domain.

8.1 Number of Balls

In the first experiment, we are predicting the total number of balls in the urn. We experiment with two different prior distributions over the number of balls: Poisson with mean 6 and uniform on the interval 1 to 8, inclusive. In both cases, each ball is black with probability 0.5. The evidence consists of color observations for 10 draws from the urn: five are black and five are white. For each prior and each observation model, five independent trials were run, each of 5 million samples.[3]

Fig. 9 shows the posterior probabilities for total numbers of balls from 1 to 15 computed in each of the five trials, along with the exact probabilities. The results

[3]Our Java implementation averages about 1700 samples per second for the exact observation case and 1300 samples per second for the noisy observation model on a 3.2 GHz Intel Pentium 4.

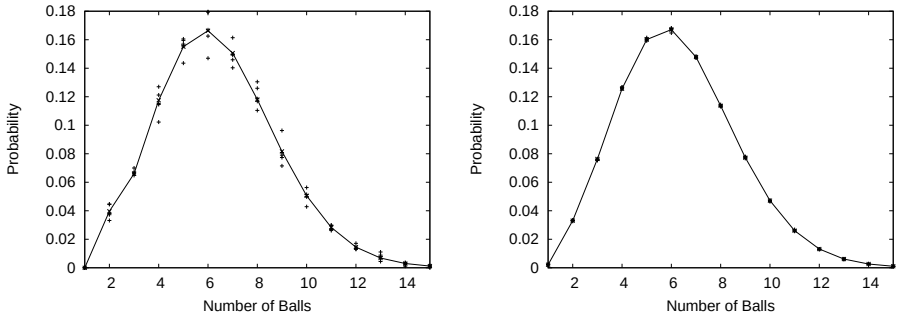

Figure 9: Posterior distributions for the total number of balls given 10 observations in the noise-free case (left) and noisy case (right). Exact probabilities are denoted by '$\times$'s and connected with a line; estimates from 5 sampling runs are marked with '+'s.

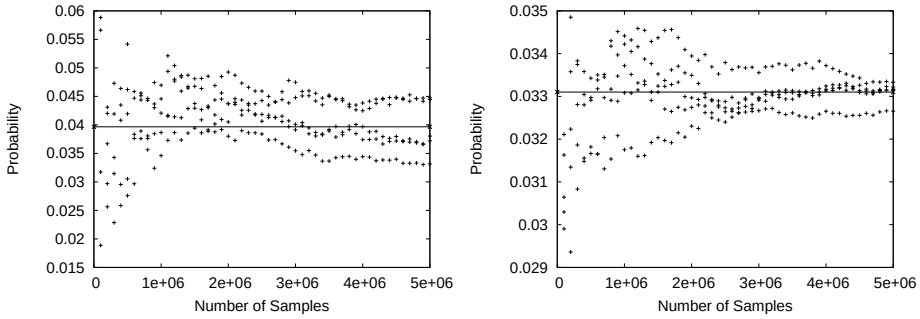

Figure 10: Probability that $N = 2$ given 10 observations (5 black, 5 white) in the noise-free case (left) and noisy case (right) for the Poisson(6) prior. Solid line indicates exact value; '+'s are values computed by 5 sampling runs at intervals of 100,000 samples.

are all quite close to the true probability, especially in the noisy-observation case. The variance is higher for the noise-free model because the sampled true colors for the balls are often inconsistent with the observed colors, so many samples have zero weights.

Fig. 10 shows how quickly our algorithm converges to the correct value for a particular probability, $P(N = 2|\text{obs})$. The run with deterministic observations stays within 0.01 of the true probability after 2 million samples. The noisy-observation run converges faster, in just 100,000 samples.

For the uniform prior, our results are shown in Figs. 11 and 12. Unlike the previous set of graphs, the prior and posterior probability distributions are dissimilar, so one might expect a likelihood weighting algorithm to perform poorly. However, we obtained comparable results.

8.1.1 Exact calculations

Before looking at the exact calculation, let us consider the generative process for a

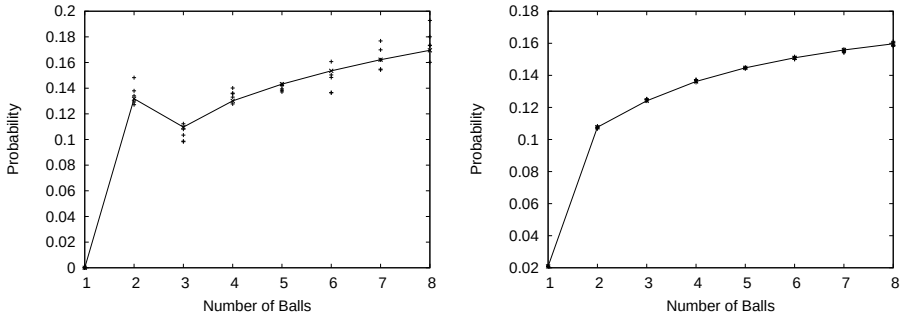

Figure 11: Posterior distributions for the total number of balls given 10 observations in the noise-free case (top) and noisy case (bottom). Exact probabilities are denoted by '$\times$'s and connected with a line; estimates from 5 sampling runs are marked with '+'s.

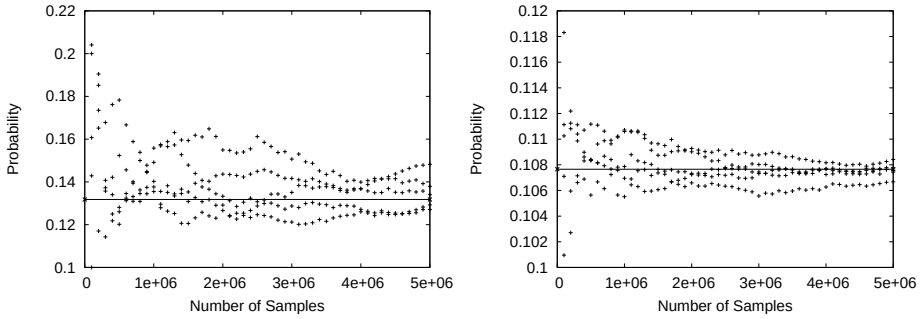

Figure 12: Probability that $N = 2$ given 10 observations (5 black, 5 white) in the noise-free case (left) and noisy case (right) for the Uniform(1, 8) prior. Solid line indicates exact value; '+'s are values computed by 5 sampling runs at intervals of 100,000 samples.

world in this model in more detail. We start constructing a possible world by sampling a value for the variable that denotes the number of balls. If we denote this variable as N, then, as pointed out in the previous subsection, $N \sim \mathsf{Uniform}(1, 8)$.

When generating the balls, we choose a color for each of them, s.t. P*(Color of the ball is black = 0.5)*. We will be interested in the probability of sampling a particular number of black and white balls given the total number of balls generated, so it is convenient to introduce random variables N_c (stands for "Number of balls of color c") for each $c \in \{b, w\}$. Clearly, $N_c \sim \mathsf{Binomial}(N, 0.5)$.

Finally, we make k draws with replacement from the urn, recording the observed colors of the balls. Note that in general, the observed color of a ball may not be the same as its true color, since the observations may be noisy. To reflect this, we will need random variables $C_1, \ldots, C_k$, each denoting the true color of the ball picked during the ith draw, and variables $O_1, \ldots, O_k$ that stand for the observed colors of the drawn balls. The probability of picking a ball of the given color on the given draw depends

on the total number of balls of that color but, given this number, is independent of outcomes of other draws, since we place the drawn ball back into the urn every time. Thus, P($C_i = c_i \mid N_{c_i} = n_{c_i}, N = n$) = $\frac{n_{c_i}}{n}$, where c_i is the true color of the ith ball drawn.

The probability of observing a particular color depends on the noise present in our observations, i.e. the probability of the observed color being different from the true color of the ball. Knowing these probabilities, we can calculate P($O_i = c$) = $\sum_{c'}$ P($O_i = c, C_i = c'$) = $\sum_{c'}$ P($O_i = c \mid C_i = c'$)P($C_i = c'$). In this formula, c' ranges over all possible colors of our "palette", and all probabilities P($O_i = c \mid C_i = c'$) are characterized by the noise parameter values. In our case, $c, c' \in \{b, w\}$. For each color c, P($O_i = c \mid C_i = c$) = 0.8 in the experiments with noisy observations and 1.0 in case of exact observations.

Now, our query in this experiment is the distribution for the random variable N, while our evidence is the particular instantiation of all the O_i variables. Letting P(x) abbreviate P($X = x$) for every variable X, and keeping in mind that N_b stands for the number of black balls, we get the following expression for P($N \mid o_1, \ldots, o_k$):

$$\mathrm{P}(n \mid o_1, \ldots, o_k) = \frac{\mathrm{P}(o_1, \ldots, o_k \mid n)\mathrm{P}(n)}{\mathrm{P}(o_1, \ldots, o_k)}$$

$$= \frac{\sum_{n_b=0}^{n} \sum_{c_1 \in \{b,w\}} \cdots \sum_{c_k \in \{b,w\}} \mathrm{P}(o_1, \ldots, o_k, c_1, \ldots, c_k, n_b)\mathrm{P}(n)}{\sum_{n'=1}^{8} \mathrm{P}(o_1, \ldots, o_k \mid n')\mathrm{P}(n')}$$

$$= \frac{\sum_{n_b=0}^{n} \binom{n}{n_b} 0.5^{n_b}(1-0.5)^{n-n_b} \prod_{i=1}^{k} (\sum_{c_i \in \{b,w\}} \mathrm{P}(o_i|c_i)\mathrm{P}(c_i))}{\sum_{n'=1}^{8} \mathrm{P}(o_1, \ldots, o_k \mid n')}$$

$$= \frac{\sum_{n_b=0}^{n} \binom{n}{n_b} 0.5^{n} \prod_{i=1}^{k} (\sum_{c_i \in \{b,w\}} \frac{n_{c_i}}{n} \mathrm{P}(o_i|c_i))}{\sum_{n'=1}^{8} \sum_{n'_b=0}^{n'} \binom{n'}{n'_b} 0.5^{n'} \prod_{i=1}^{k} \sum_{c'_i \in \{b,w\}} \frac{n_{c'_i}}{n} \mathrm{P}(o_i|c'_i)}$$

8.2 Identity Uncertainty

In the second experiment, three balls are drawn from the urn: a black one and then two white ones. We wish to find the probability that the second and third draws produced the same ball. The prior distribution over the number of balls is Poisson(6). Unlike the previous experiment, each ball is black with probability 0.3.

We ran five independent trials of 100,000 samples on the deterministic and noisy observation models. Fig. 13 shows the estimates from all five trials approaching the true probability as the number of samples increases. Note that again, the approximations for the noisy observation model converge more quickly. The noise-free case stays within 0.01 of the true probability after 70,000 samples, while the noisy case converges within 10,000 samples. Thus, we perform inference over a model with an unbounded number of objects and get reasonable approximations in finite time.

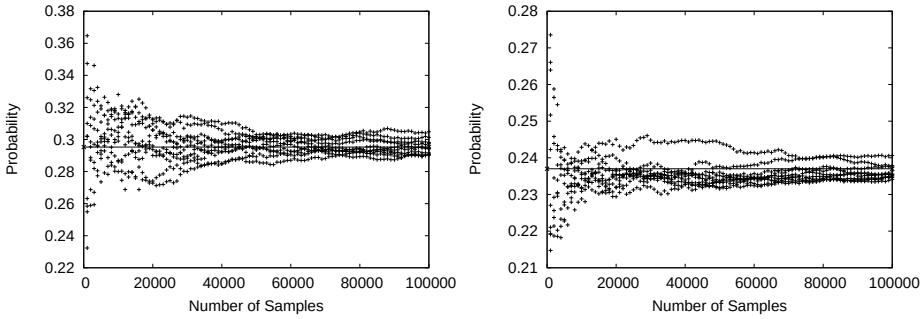

Figure 13: Probability that draws two and three produced the same ball for noise-free observations (left) and noisy observations (right) of the identity uncertainty experiment. Solid line indicates exact value; '+'s are values computed by 5 sampling runs.

8.2.1 Exact calculations

Similarly in spirit to calculations of section 8.1.1 and following the same conventions for the variable names, we represent the question whether the ball chosen at the i-th draw was the same as the one drawn on the j-th draw by a query to the Boolean random variable S_{ij}, thereby obtaining

$$\mathrm{P}(s_{ij} \mid o_1, \ldots, o_k) = \frac{\mathrm{P}(o_1, \ldots o_k, s_{ij})}{\mathrm{P}(o_1, \ldots, o_k)}$$

However, this time we need to restrict the balls at ith and jth draws to be the same. With no available evidence, the chance of this happening in any given world is P'$(s_{ij} \mid n_b, n) = \sum_{c_{ij} \in \{b,w\}} \frac{n_{c_{ij}}}{n} \frac{1}{n}$, because the chosen ball may be either black or white. When we have observations, we need to incorporate them into our estimate. Noisy evidence makes it possible to observe different colors for the two draws even if we picked the same ball during both. Therefore, the probability of picking the same ball of color c in a fixed world twice given the observations o_i and o_j is $\frac{n_{c_{ij}}}{n} \frac{1}{n} \mathrm{P}(o_i \mid c) \mathrm{P}(o_j \mid c)$. To get the probability of picking the same ball of any color, $\mathrm{P}(s_{ij} \mid n_b, n)$ we simply sum over the colors. The complete derivation takes the following form:

$$\mathrm{P}(s_{ij} \mid o_1, \ldots, o_k) = \frac{\mathrm{P}(o_1, \ldots o_k, s_{ij})}{\mathrm{P}(o_1, \ldots, o_k)}$$

$$= \frac{\sum_{n=1}^{\infty} \sum_{n_b=0}^{n} \sum_{c_1 \in \{b,w\}} \cdots \sum_{c_k \in \{b,w\}} \mathrm{P}(o_1, \ldots, o_k, c_1, \ldots, c_k, n_b, n, s_{ij})}{\sum_{n'=1}^{\infty} \mathrm{P}(o_1, \ldots, o_k \mid n') \mathrm{P}(n')}$$

$$= \frac{\sum_{n=1}^{\infty} \sum_{n_b=0}^{n} \mathrm{P}(n_b, n) (\prod_{t=1, t \neq i,j}^{k} \sum_{c_t \in \{b,w\}} \frac{n_{c_t}}{n} \mathrm{P}(o_t | c_t)) \mathrm{P}(s_{ij} | n_b, n)}{\sum_{n'=1}^{\infty} \sum_{n'_b=0}^{n'} \mathrm{P}(n'_b, n') \prod_{s=1}^{k} \sum_{c'_s \in \{b,w\}} \frac{n_{c'_s}}{n'} \mathrm{P}(o_s | c'_s)},$$

where

$$\mathrm{P}(s_{ij}|n_b, n) = \sum_{c_{ij} \in \{b,w\}} \frac{n_{c_{ij}}}{n} \frac{1}{n} \mathrm{P}(o_i \mid c_{ij}) \mathrm{P}(o_j \mid c_{ij})$$

and

$$\mathrm{P}(n_b, n) = \binom{n}{n_b} 0.5^n \mathrm{P}(n).$$

The infinite summations over the values of n are hard to compute analytically, so in practice, we calculated it out to the 170th term.[4]

9 Related Work

Researchers engaged in probabilistic modelling for various applications have had to define distributions over worlds with unknown objects. Examples include record linkage (Fellegi & Sunter, 1969) and aircraft tracking (Reid, 1979). There have also been several recent proposals for general representations for relational modeling. BLOG's clausal syntax is inspired by BLPs (Kersting & De Raedt, 2001), although BLPs only define distributions over Herbrand models. Basic PRMs (Koller & Pfeffer, 1998) have been extended in various ways, including *number uncertainty*—uncertainty about the number of objects standing in some relation to a single existing object—and *existence uncertainty*, such as uncertainty about whether there is a role for a given actor in a given movie (Getoor et al., 2002). Features have also been added to PRMs for modeling uncertainty about the total number of objects of a given type (Pasula & Russell, 2001). BLOG's number statements subsume these three types of uncertainty in an elegant way. BLOG also improves on PRMs by allowing functions of arbitrary arity, making it easy to define variables such as the state of an aircraft at a given time point. Another line of work represents relational and domain uncertainty using undirected or feature-based models. Examples include (McCallum & Wellner, 2003), (Taskar et al., 2002), and (Domingos & Richarson, 2004). Finally, the language of Bayesian networks with plates (Gilks et al., 1994) has been extended recently to handle context-specific and recursive dependencies (Mjolsness, 2004), but plate models still represent distributions over joint instantiations of random variables, not over relational structures.

10 Conclusions

Author: Brian

[4] This limitation is imposed by the precision of the double floating point number in Java. However, the final probability is accurate to 10^{-15} long before the 170th term in the summation is added.

A Syntax Reference

A.1 Syntactic conventions and reserved words.

Identifiers
BLOG identifiers (type, functor, object, and variable names) can be any sequence of alphanumeric and underscore characters starting with a letter, modulo the reserved words. By convention (not enforced), type, object, and functor names start with an upper-case letter while variable names start with a lower-case letter.

Reserved words
They are: EXISTS, else, elseif, false, for, FORALL, generating, guaranteed, if, non-random, null, random, then, true, truth, type, value.

The words true and false must start with a lower-case character. For the rest of the reserved words, the shown capitalization is conventional though not enforced. The reserved words should not be used as identifiers.

Alphanumeric entities
BLOG supports:

- integers as sequences of digits of the form 12345 (these consitute built-in type NaturalNum);
- doubles in the format 1234.5678; any double must contain at least one digit in its integer part and at least one digit in its fractional part (encoded as built-in type Real);
- vectors in the format [num_1, ..., num_n], where num_i is an NaturalNum or a Real; each vector is an object of type RkVector for some $k \geq 2$;
- strings as arbitrary sequences of characters and blanks enclosed in " "(as objects of BLOG type String) ;

Logical connectives
BLOG uses the following symbols for the logical connectives:
& for conjunction;
| for disjunction;
! for negation;
$\rightarrow$ for implication.
The equality test is supported for logical terms by the '=' sign.

B Technicalities of Semantics

B.1 Event space over possible worlds

Define event space (σ-field) over Ω_M.

Show that terms, formulas, and set expressions are all measurable functions with respect to this event space.

References

Domingos, P., & Richarson, M. (2004). Markov logic: A unifying framework for statistical relational learning. *Proc. ICML Wksp on Statistical Relational Learning and Its Connections to Other Fields*.

Enderton, H. B. (2001). *A mathematical introduction to logic*. Academic Press. 2nd edition.

Fellegi, I., & Sunter, A. (1969). A theory for record linkage. *JASA, 64*, 1183–1210.

Friedman, N., Getoor, L., Koller, D., & Pfeffer, A. (1999). Learning probabilistic relational models. *Proc. 16th IJCAI* (pp. 1300–1307).

Getoor, L., Friedman, N., Koller, D., & Taskar, B. (2002). Learning probabilistic models of link structure. *JMLR, 3*, 679–707.

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

Kersting, K., & De Raedt, L. (2001). Adaptive Bayesian logic programs. *Proc. 11th Int'l Conf. on ILP*.

Koller, D., & Pfeffer, A. (1998). Probabilistic frame-based systems. *Proc. 15th AAAI* (pp. 580–587).

McCallum, A., & Wellner, B. (2003). Toward conditional models of identity uncertainty with application to proper noun coreference. *IJCAI Wksp on Information Integration on the Web*.

Milch, B., Marthi, B., & Russell, S. (2004). BLOG: Relational modeling with unknown objects. *ICML Wksp on Statistical Relational Learning*. Banff, Alberta, Canada.

Milch, B., Marthi, B., Sontag, D., Russell, S., Ong, D. L., & Kolobov, A. (2005). Approximate inference for infinite contingent Baysian networks. *10th Int'l Wksp on Artificial Intelligence and Statistics*.

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

Pasula, H. (2003). *Identity uncertainty*. Doctoral dissertation, UC Berkeley.

Pasula, H., Marthi, B., Milch, B., Russell, S., & Shpitser, I. (2003). Identity uncertainty and citation matching. In *Nips 15*. Cambridge, MA: MIT Press.

Pasula, H., & Russell, S. (2001). Approximate inference for first-order probabilistic languages. *Proc. 17th IJCAI* (pp. 741–748).

Pearl, J. (1988). *Probabilistic reasoning in intelligence systems*. San Francisco: Morgan Kaufmann. Revised edition.

Reid, D. B. (1979). An algorithm for tracking multiple targets. *IEEE Trans. on Automatic Control, 6*, 843–854.

Russell, S. (2001). Identity uncertainty. *Proc. 9th Int'l Fuzzy Systems Assoc. World Congress*.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach*. Morgan Kaufmann. 2nd edition.

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

Skolem, T. (1928). Über die mathematische Logik. *Norsk Matematisk Tidsskrift, 10*, 125–142.

Taskar, B., Abbeel, P., & Koller, D. (2002). Discriminative probabilistic models for relational data. *Proc. 18th UAI* (pp. 485–492).

Wellner, B., McCallum, A., Peng, F., & Hay, M. (2004). An integrated, conditional model of information extraction and coreference with application to citation matching. *Proc. 20th UAI*.