The BLOG Language for Probability Models version 0.6

Lei Li
Computer Science Division
University of California Berkeley
leili@cs.berkeley.edu

Stuart Russell
Computer Science Division
University of California Berkeley
russell@cs.berkeley.edu

April 9, 2013

Abstract

This document introduces the syntax of BLOG, a probabilistic programming language, to describe random variables and their probability dependencies. BLOG defines probabilistic generative models over first-order structures. For example, all Bayesian networks can be easily described by BLOG. BLOG has the following features: (a) it employs an open-universe semantics; (b) it can describe relational uncertainty; (c) it can handle identity uncertainty; (d) it is empowered by first-order logic. The particular syntax covered in this document corresponds to BLOG version 0.6. The current version includes significant redesign and extension to the previous versions of BLOG, based on the principles of usability and implementation efficiency.

Contents

1 Summary of changes since version 0.3 (2008)

- Support of Map data structure;
- Categorical Distribution and TabularCPD using Maps;
- Uniform argument representation in CPD;
- Nested if-then-else in dependency statement;
- Direct support of boolean operations;

- Arithmetic operators (+, -, *, /, ^), which can be used in both fixed and random function statements;
- Support of Array, reference to array elements, and basic matrix operators on Real Array (+, -, *);
- All internal operators are type checked;
- Support additional operators and build-in functions;
- Without causing confusion, "nonrandom" is changed to "fixed", "guaranteed" is changed to "distinct";
- Complete redesign of parser and semantic translator, towards parsing efficiency and modularity.

2 Basic language concepts

A BLOG program consists a list of statements. Each statement ends with semicolon(;). Statements include

1. Type declaration;
2. Fixed function declaration;
3. Distinct symbol declaration;
4. Random function declaration;
5. Origin function declaration;
6. Number statement;
7. Evidence statement;
8. Query statement.

BLOG is strongly typed, therefore every variable and function should explicitly specify a type. Each BLOG program defines a set of random variables and their probabilistic dependencies. A very toy example of defining a random variable in BLOG is:

```
random Real x ~ Gaussian(0, 1);
```

which states that a real-valued random variable x is distributed according to the standard normal distribution. To specify a dependent variable $y|x \sim \mathcal{N}(x,1)$:

```
random Real y ~ Gaussian(x, 1);
```

3 Declaring types

BLOG is a strongly typed language. To declare a type in BLOG:

```
type typename;
```

There are built-in types including: `Boolean`, `Integer`, `Real`, `String`, and array. There are *literals* in built-in types, e.g. `1.0`, `"abc"`, true, false.

4 Fixed functions

A fixed function has the same semantic interpretation, i.e. their values do not change over possible worlds. To declare a function with fixed interpretation for all satisfying possible worlds:

```
fixed type0 funcname(type1) = expression;
```

The statement defines a function with name `funcname` with one argument of type, `type1`, and return type, `type0`.

The function body is an `expression`, which can be

- a literal of built-in types;
- expression such as $a+b$, (see later sections);
- A function call to external interpretation in Java-class, with passing of fixed term as arguments.

Functions can have zero or multiple arguments as well. Functions without arguments are constants.

4.1 Constants

Constant is a zero-ary fixed function.

```
fixed typename name = expression;
```

Where `expression` should not contain any free variables.

For example:

```
fixed Real a = 1.0;
fixed Boolean b = true;
```

Another example involves a bit more on expression.

```
fixed Real c = 1.0 + a;
```

Such names can be referred anywhere fixed zero-ary function could appear. For example:

```
random Integer x ~ Poisson(a);
```

Here x follows a Poisson distribution with the parameter set to a.

4.2 Distinct "object" symbols

There is a special type of functions, distinct symbols. Distinct symbols are fixed zero-ary functions without function bodies.

```
distinct typename name1, name2, name3, ...;
```

It defines several symbols of name, `name1`, `name2`, `name3` .... These symbols will have fixed interpretation in all satisfying possible worlds. In addition, all these symbols will have *different* but fixed interpretations. We can have multiple `distinct` statements in one blog, and all those symbols for the same type will have distinct interpretation in possible worlds. In this sense, these symbols are equivalent to "objects" in model structures.

The following BLOG code declares 100 Person symbols.

```
type Person;
distinct Person P[100];
```

Build-in distinct symbols There are predefined distinct symbols for Boolean, Integer, Real, and String, including all integers, all real numbers, and text strings, e.g. 1, 3.14, "hello". These predefined symbols are also called *literals*.

5 Random functions

Random functions may have different interpretations across possible worlds. Random functions are defined in the similar way as fixed functions, but with `random` keyword. To declare a random function:

```
random type0 funcname(type1 x) dependency-expression;
```

The statement defines a random function with name `funcname` with one argument of type, `type1`, and return type, `type0`.

As a very simple example, we can declare the height of a Person with the following code.

```
type Person;
distinct Person Alice, Bob;
random Real height(Person p) ~ Gaussian(1.70, 0.25);
```

6 Origin functions

Origin functions are used in number statement to generate objects in possible worlds.

```
origin type0 funcname(type1);
```

An origin function has exactly one argument type and one return type. The meaning of origin functions will be more clear in number statements.

7 Number statement

BLOG supports open worlds semantics, i.e. the number of objects in possible worlds can be declared in the language itself. While traditional graphical models does not support such semantics. For user declared type, the number statements specify how many objects there are in each type, and how they are generated.

```
#typename dependency-expression;
#typename(origin_function=x, ...) dependency-expression;
```

For example, the following example declares the number of `Person` follows Poisson distribution.

```
type Person;
#Person ~ Poisson(10.0);
```

Another example of number statement with origin function. It declares that the number of aircraft follows Poisson distribution, each Blip has a Source object, and the number of Blips generated by an Aircraft follows Bernoulli distribution.

```
type Aircraft;
type Blip;
#Aircraft ~ Poisson(10.0);
origin Aircraft Source(Blip);
#Blip(Source=a) ~ Bernoulli(0.5);
```

8 Dependency statement

In both nonrandom function declarations and number statements, the main body consists of dependency statements, which specifies generation process. A dependency statement can be one of the following form: simple distribution clause, simple operator clause, or conditional clauses. A distribution clause is $\sim$ followed by the distribution name and arguments. Arguments can be any expressions allowed by the distribution.

```
~ Distribution(args)
```

For example, here is code to define values from Poisson distribution.

```
~ Poisson(10.5)
```

Operator clause is specified as

```
= expression
```

where `expression` is arithmetic or relational operation.

Conditional clauses uses `if then else`,

```
if cond then clause1
else clause2
```

where `cond` is a Boolean valued expression, and `clause1`, `caluse2` can be one of three clauses: simple distribution clause, operator clause or conditional clause.

Example 1 (Uneven coin)**.** *There are two coins, one evenly weighted and one skewed. However, there is no visually distinction between the two. Each time we pick a coin, throw and check the face.*

```
random Boolean even()
  ~ Bernoulli(0.5);
random Boolean head()
  if even() == EvenCoin then
    ~ Bernoulli(0.5)
  else
    ~ Bernoulli(0.8);
```

9 Observing evidence

There are two types of evidence statements.

```
obs expression1 = expression2;
```

where `expression1` should be random function application expression without free variables.

For example:

```
random Real x ~ Gaussian(1.0);
obs x = 0.5;
```

10 Issuing queries

```
query expression;
```

where `expression` is a random function application expression without free variables. The result will be the posterior distribution given the observations.

11 Expressions

An expression can include both nonrandom and random terms. It is defined as

- a reference to literals of Builtin types, e.g. Integer, Real, String, Boolean and Timestep.
- a reference to distinct symbols;
- a reference to constant symbols;
- a proper reference to an element in `Array`, with index of general expression;
- fixed, random or origin function with a *expression* as arguments `randomfun(p1, p2, ...)`, where `p1, p2, ...` are also *general expressions*;
- arithmetic operation on numerical type: `e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2`, `- e1`, `+ e1`, `- e1`, `(e1)`, where `e1` and `e2` are also *expression* of type `Integer` or `Real`.
- logic expression on Boolean type: `e1 & e2`, `e1 | e2`, `! e1`, `(e1)` where `e1` and `e2` are also *expression* of type `Boolean`.
- relational expression: `e1 > e2`, `e1 >= e2`, `e1 < e2`, `e1 <= e2`, where `e1` and `e2` should be general expression with values comparable; `e1 == e2`, `e1 != e2` can be any general expression.

11.1 Fixed expression

Expression without any appearance of random function symbols. Example:

```
1.0 + 2.0 * 3.0
a - 2.0
Twice(10.0) * 5.5
```

Where `Twice(·)` is declared as

```
fixed Real Twice(Real x) = x * 2;
```

11.2 Quantified formula

BLOG allows quantified formula as in first-order logic. To specify a universal quantified formula,

```
forall typename x expression
```

To specify an existential quantified formula,

```
exists typename x expression
```

11.3 Set expression

Set expression is a special type of express which can only be used as arguments in function call, and observed symbol evidence.

```
{typename x:condition(x)}
```

For example, to specify uniform choice from all balls,

```
type Ball;
#Ball ~ Poisson(10.0);
random Ball choice() ~ UniformChoice({Ball b});
```

To specify symbol evidence,

```
obs {Ball b} = {B1, B2, B3};
```

12 Array type

to declare an array type

```
type[]
```

12.1 Constant array

To declare an constant array

```
fixed type[] name = List_literal;
```

For example, to declare an array with natural numbers:

```
fixed NaturalNum[] c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

When referring to an element in the array, it could be referred as `c[0], c[1], c[2]`, etc.

12.2 Constant list literals

As we already seen, we could use the square brackets, `[]` to denote constant list literals. Elements in a list is separated by a comma (,). Lists can be nested within a list. A shorthand notation is to use semicolon (;) to separated multiple lists within a list. The following two lists are regarded equivalent.

```
[1, 2, 3; 4, 5, 6];
[[1, 2, 3], [4, 5, 6]];
```

Constant list literals are used to assign values to an array or passing parameters to a function.

12.3 Multi-dimensional array

Since `Array` is a type, itself could be nested in Array declaration, thus yielding multi-dimensional array.

```
fixed type[][] table = [...];
```

For example, a two dimensional array of int will be

```
fixed NaturalNum[][] table = [[1, 2, 3], [4, 5, 6]];
```

The following syntax in short hand is also correct:

```
fixed NaturalNum[][] table = [1, 2, 3; 4, 5, 6];
```

For example, a transition matrix in Kalman filters with Newton dynamics can be declared as:

```
fixed Real[][] A = [1, 1, .5; 0, 1, 1; 0, 0, 1];
```

An element in such a dimensional array can be referred as `A[0][0]`.

12.4 Size of an array

There are two special functions obtain the size of an array. `length(·)` and `size(·)`. `length(·)` returns the length of one dimensional array, while `size(·)` returns the lengths at all dimensions. In the above example, `length(table)` will be 3, and `size(table)` will be `[2, 3]`.

13 Map

Map are specified using braces.

```
{key1 -> value1, key2 -> value2}
```

For example,

```
 {true -> 0.3, false -> 0.7}
```

A Map's key must be some constant, while its value can be evaluated value of a *expression*, as long as the type matches.

```
{true -> x^2, false -> y/2}
```

In addition, `type2` in a map can be the `Distribution` type, which will be introduced in Section 14.

13.1 Multi-dimensional map

The type in a map can be an array, which will result in a multi-dimensional map. For example,

```
{[1, 1] -> 0.1, [1, 2] -> 0.3, [2, 1] -> 0.2, [2, 2] -> 0.4};
```

This will be useful in creating TabularCPD (see later sections) with multiple parent variables.

14 Probability Distribution Library

14.1 Elementary Distribution

Currently, many distributions are supported by BLOG. A full list of distributions are included in the appendix.

For example, Gaussian distribution can be referred as,

```
Gaussian(Real, Real)
```

14.2 Categorical distribution as defined by probability mass table

```
Categorical(Map_expression);
```

It defines probability mass using a map.

The probability should sum up to 1.0, otherwise it will by default add an entry null -> residual probability. For example:

```
Categorical({true -> 0.3, false -> 0.7});
```

If the probabilities sum to more than 1.0, the BLOG compiler will produce a runtime error.

14.3 TabularCPD

To declare and construct a tabular conditional probability distribution,

```
  TabularCPD(Map<type2, Distribution<type1>>, v);
```

For example:

```
TabularCPD({true -> ~ Bernoulli(0.3),
            false -> ~ Bernoulli(0.6)}, x);
```

With the comprehension, we could even declare conditional mixture of Gaussian very easily. For example:

```
random Int z ~ Categorical({0 -> 0.4, 1 -> 0.6});
random Real x ~ TabularCPD({0 -> ~Gaussian(5, 1.0),
                 1 -> ~Gaussian(10, 1.0)}, z);
```

14.3.1 Multiple dependent variables

If a CPD is dependent on several parent variables, a multi-dimensional map will be used

```
TabularCPD({[0, 0] -> ~Gaussian(5, 1.0),
            [0, 1] -> ~Gaussian(10, 1.0),
            [1, 0] -> ~Gaussian(2, 4.0),
            [1, 1] -> ~Gaussian(20, 4.0)}, [x, y])
```

15 Extending BLOG

15.1 User defined distribution

Probability distributions are implemented in Java and should implement the interface `blog.distrib.CondPro`
Alternatively, it could be declared as subclass of `blog.distrib.AbstractCondProbDistrib`.
By default, BLOG engine will look up distribution classes in the package `blog.distrib`. In addition, it will look up distribution classes under the default empty package.

Note: using one distribution class to implement deterministic operations are supported but not recommended.

Below is one example of uniform distribution on Integer.

```
import java.util.*;
import blog.*;
import blog.distrib.*;
import blog.common.Util;
import blog.model.Type;

public class UniformInt extends AbstractCondProbDistrib {
  public UniformInt(List params) {
    try {
      lower = ((Number) params.get(0)).intValue();
```

```
      upper = ((Number) params.get(1)).intValue();
      if ((lower > upper) || (params.size() > 2)) {
        throw new IllegalArgumentException();
      }
    } catch (RuntimeException e) {
      throw new IllegalArgumentException(
          "UniformInt CPD expects two integer arguments "
              + "[lower, upper] with lower <= upper.  Got: " + params);
    }
  }

  public double getProb(List args, Object value) {
    if (!args.isEmpty()) {
      throw new IllegalArgumentException(
          "UniformInt CPD does not take any arguments.");
    }
    if (!(value instanceof Integer)) {
      throw new IllegalArgumentException(
          "UniformInt CPD defines distribution over objects of class "
              + "Integer, not " + value.getClass() + ".");
    }
    int x = ((Integer) value).intValue();

    if ((x >= lower) && (x <= upper)) {
      return 1.0 / (upper - lower + 1);
    }
    return 0;
  }

  public Object sampleVal(List args, Type childType) {
    if (!args.isEmpty()) {
      throw new IllegalArgumentException(
          "UniformInt CPD does not take any arguments.");
    }

    double x = lower + Math.floor(Util.random() * (upper - lower + 1));
    return new Integer((int) x);
  }

  private int lower;
  private int upper;
}
```

A Built-in operators and functions

Table 1: Arithmetic operators on Integer and Real

operator	interpretation
+	plus
−	minus
*	multiply
/	divide
%	module (only applied to Integers)
^	power

Table 2: Logical operators on Boolean

operator	interpretation
&	and
|	or
!	not
=>	imply

Table 3: Quantified formula

operator	interpretation
forall	∀
exists	∃

B Built-in distributions

- Bernoulli
- Beta
- Binomial
- Categorical
- Exponential
- Gamma

Table 4: Relational operators on Integer, Real and other comparable types

operator	interpretation
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Table 5: Equality operator on all types

operator	interpretation
==	equal to
!=	unequal to

- Gaussian
- Geometric
- NegativeBinomial
- Poisson
- TabularCPD
- UniformChoice
- UniformInt
- UniformReal
- MultivarGaussian
- Dirichlet

Table 6: Operators on String

operator	interpretation
+	concatenate
==	equal to
==	unequal to
`IsEmptyString()`	returns True if the string is empty

Table 7: Operators on Timestep

operator	interpretation
`Pred()`	previous Timestep