BFiT: From Possible-World Semantics to Random-Evaluation Semantics in Open Universe

Yi Wu *
Computer Science Division
UC Berkeley

Lei Li †
Baidu Research

Stuart Russell*
Computer Science Division
UC Berkeley

Abstract

In recent years, several probabilistic programming languages (PPLs) have emerged, such as Bayesian Logic (BLOG), Church, and Figaro. These languages can be classified into two categories: PPLs interpreted using possible-world semantics and ones using random-evaluation semantics. In this paper, we explicitly analyze the equivalence between these two semantics in the context of open-universe probability models (OUPMs). We propose a novel *dynamic memoization* technique to construct OUPMs using procedural instructions in random-evaluation based PPLs. We implemented a translator named BFiT, which converts code in BLOG (possible-world based) to Figaro (random-evaluation based). The translated program in Figaro exhibits a merely constant blowup factor in program size while yielding the same inference results as the original model in BLOG.

1 Introduction

In *possible-world semantics*, a probabilistic program defines a probability measure over sets of *possible worlds*. Syntactic symbols in the program are interpreted as random variables which have concrete values in each possible world. Languages such as Bayesian Logic (BLOG) [4] and Markov Logic Network (MLN)[6] fall into this category. In this paper, we are interested in PPLs for generative models. Therefore we will illustrate our ideas using BLOG.

In contrast, a program in probabilistic programming languages such as Church [2] and Figaro [5] defines a probability measure over execution traces or partial traces. The semantics of this category is called *random-evaluation semantics* or *random-execution semantics*.

A related perspective is from the view of the taxonomy of domain specific languages (DSLs).

Independent versus embedded: PPLs adopting possible-world semantics are often implemented as independent DSLs with their own syntax. While those with random-evaluation semantics are often embedded in general-purpose programming languages: Church is embedded in Scheme; Figaro is embedded in Scala.

Declarative versus procedural: Possible-world based PPLs are *declarative*. While random-evaluation based PPLs are often *procedural* (corresponding to the execution traces).

In regard to expressiveness, PPLs have different capabilities of describing probabilistic models. Open-universe probability model (OUPM) is one of the most powerful models, and proves to be very useful for practical applications, such as Seismic monitoring [1]. OUPMs model uncertainties not only in values of random variables, but also in the existence and identity of objects as well as the relations between them. Two fundamental capabilities of OUPMs are the expressive power of (a) context-specific dependency and (b) (possibly recursive) random number of objects. Syntactically,

*{jxwuyi,russell}@cs.berkeley.edu
†lilei22@baidu.com

```
random Boolean X ~ BooleanDistr(0.5);
random Boolean Y ~
  if X then BooleanDistr(0.9);
  else BooleanDistr(0.1);
```

Figure 1: a simple example in BLOG

```
def X():Element[Boolean] = Flip(0.5);
def Y():Element[Boolean] =
     If(X,Flip(0.9),Flip(0.1));
```

Figure 2: a simple example in Figaro

```
type Ball;
#Ball ~ Poisson(5);
```

Figure 3: an open-universe example in BLOG

```
class Ball;
val all_balls:Element[List[Ball]] =
  MakeList(Poisson(5),()=>Select(1.0->new Ball));
```

Figure 4: an open-universe example in Figaro

the number of random variables in an OUPM is unbounded, and dependencies between them can be even cyclic[1].

So far, BLOG is the only PPL that can leverage the full expressive power of OUPMs. In this work, we explore possible ways to translate a program describing OUPMs from BLOG to PPLs based on random-evaluation semantics. As described above, random-evaluation based PPLs are inherently procedural and embedded in general-purpose programming languages, referred as host languages. Our translation task is essentially to preserve possible-world semantics through procedural instructions. Prior work has proposed *memoization* as a general technique to maintain possible-world semantics in a procedural language [3]. However, straightforward memoization is inadequate for translating programs modeling OUPMs, as embedded DSLs (e.g, Church, Figaro) are limited in their internal handling of the number of random variables or objects in OUPMs. Translation typically must rely on a host language (e.g., Scala) to construct data structures for possible worlds, maintain them, and map them to equivalent evaluation traces in the target PPL (e.g., Figaro).

This paper advances the field of PPL research with the following contributions. We propose a general technique called *dynamic memoization* to construct programs mimicking possible-world semantics in procedural PPLs based on random-evaluation semantics. We apply our technique to Figaro, and develop a Blog-to-Figaro-Translator (BFiT) to automatically translate a model written in BLOG to a Figaro program. The transformed program yields the same inference results while keeping a provable constant blowup factor in program size.

This work shows some insights of the essential differences between PPLs adopting different semantics and brings the theoretical idea [3] into practice. Furthermore, BFiT illustrates the possibility for a PPL having very concise syntax to fully take the computational power of another PPL embedded in a particular general-purpose programming language.

2 Overview of BLOG and Figaro

In this section, we present a brief tutorial on BLOG and Figaro.

In Figure 1 and 2, we demonstrate the programs modeling the same Bayesian network written in BLOG and Figaro respectively. In this Bayesian network, there are two Boolean random variables `X` and `Y`, where the probability of `Y` becoming true is depending on the value of `X`.

The syntax of BLOG dependency statement is consistent with ordinary mathematical notations — references to the syntactically same symbol always correspond to the same random variable, therefore they have the same value in a concrete possible world. In Figaro, for each random variable, one defines a Scala function using the key word `def` [2] with a special return type of `Element[Boolean]`. `Element[]` is a special template class in Scala provided by Figaro to represent a random variable in the underlying Bayesian network. When an object of type `Element[]` is generated, a random variable will be introduced to the underlying Bayesian network. Hence, whenever the function `X` or `Y` in Figure 2 is evaluated, a new random variable will be generated correspondingly. Consequently if we make a query of $Y \vee \neg Y$ in the BLOG model in Figure 1, the answer is always true. By contrast, if we evaluate the same formula using the program in Figure 2, the answer varies.

[1] For concrete examples, please download BLOG and refer to the Hurricane model and PCFG model.

[2] One familiar with Figaro might point out that a more straightforward approach to define random variables is to use the key word `val`. However, using `val` is not general enough to handle models with cyclic dependency or unknown number of random variables. For concrete examples, please download BLOG and refer to the Hurricane model and the SybilAttack model.

```
type Person, Login;      #Person ~ Poisson(5);
random Boolean Honest(Person x) ~ BooleanDistrib(0.9);
origin Person Owner(Login);
#Login(Owner = x) ~ if Honest(x) then 1 else Geometric(0.8);
random Login sample ~ UniformChoice({l for Login l});
query Honest(Owner(sample));
```

Figure 5: Person-Login model in BLOG

We also demonstrate programs describing a simple open-universe probabilistic model written in BLOG (Figure 3) and Figaro (Figure 4) respectively. In this model, there is a type Ball, and the total number of balls obeys a Poisson distribution. In BLOG, the total number of balls, denoted by `#Ball`, is treated as a random variable. We call this special way of defining a number variable a *Number Statement*. The generation of all objects of type Ball will be handled automatically by the back-end system of BLOG. However, in Figaro, we need to define a Scala class `Ball` to represent the type and invoke a special Figaro function `MakeList` to generate a list, whose length obeys a Poisson distribution, containing all the generated objects of Scala class `Ball`.

3 Case Study

In this section, we demonstrate a concrete model written in BLOG, namely the Person-Login model[3]. Based on this model, we introduce the key idea of *dynamic memoization* in our Blog-to-Figaro-Translator, BFiT.

The program describing the Person-Login model is shown in Figure 5. In this program, there are two kinds of special statements on line 3 and line 4. The statement on line 3 defines an *origin function*. This indicates that the existence of an object of type Login is depending on some particular object of type Person, which is represented by the *origin function* `Owner(·)`. The statement on line 4 is a special variant of number statement. We call it a *number statement with origin functions*. It defines the distribution of the total number of objects of type Login whose origin functions, `Owner(·)`, refer to the same object.

In the Person-Login model, there are two types, `Person` and `Login`. The number of different persons obeys a Poisson distribution. For each person, there is a random variable `Honest(·)` indicating whether this person is honest or not. An honest person will exactly have a single login while a dishonest person might have multiple logins, whose total number obeys a Geometric distribution. Finally, we would like to query the honesty of the owner of a randomly sampled login.

In the Person-Login model, the total number of different logins is contingent on the objects of type Person and the associated random variables, `Honest(·)`. Furthermore, the query requires access to all the different logins to draw a sample from. For BLOG, the back-end system automatically maintains all the generated objects, which makes it straightforward for a user to get access to some particular set of generated objects. However, in order to describe this model in Figaro, we have to explicitly design a data structure that could be used for both efficient object generation and storage.

Note that for each type in BLOG, we need to correspondingly define a Scala class with the same name in Figaro to represent that type. There are two possible ways to *dynamically memoize*. One idea here is that whenever an object of class `Person` is generated, we correspondingly generate all the affiliated objects of `Login` and store them all together inside the class body of `Person`. When required to get access to all the logins, we enumerate all the objects of person and then aggregate all the affiliated logins stored inside the class body.

Another idea comes from the fact that the dependency between Person and Login is acyclic. So we could firstly generate all the objects of `Person` and store them in a list. Afterwards, based on this list, we could generate a new list containing all the objects of `Login`.

The first idea is more general and straightforward but requires a very expensive aggregation operation when accessing all the objects of some type. The second idea is more efficient but requires acyclic dependency between types. For simplicity, BFiT adopts the second idea and does not handle models with cyclic type dependency in the current version.

[3]This is a simplified version of SybilAttack model, which can be downloaded from BLOG's website.

Table 1: lines of code for 7 models in BLOG and the target code in Figaro

Model	CSI	Burglary	Hurricane	UrnBall	Citation	SybilAttack	TugWar
LoC in BLOG	14	22	37	38	40	14	55
LoC in Figaro	71	87	168	126	178	83	278

4 Proposed Method

BFiT applies the following techniques for *dynamic memoization*.

Type and random function: For each type declaration or random function in BLOG, we define a corresponding Scala class or Scala function with the same name in Figaro.

Moreover, for each function we define in Scala, we create a hash map used for memoization. Thus when a function is re-evaluated with the same input arguments, it always refers to the same object of type `Element[]`, namely the same random variable defined in the underlying Bayesian network.

Number statement and object storage: For each type in BLOG, we also define a special random variable in Figaro to represent the number statement, which is memoized as well. This random variable will be defined before evaluating any random function related to the corresponding type. Moreover, we use a *List* data structure in Scala to store all the objects for each type. The object generation will be done at the same time as the definition of the special random variable representing number statement.

Origin functions: For each origin function in the form of `origin A func(C)` in BLOG, we treat `func` as a field of type `A` in the body of `class C` in Scala since for each object of `C`, it is associated with a fixed parent, namely a particular object of type `A`.

Note that origin function introduces type dependency. A topological sort over the type dependency will be performed by BFiT. Object generation and storage will be processed following the resulting topological order.

In the current version of BFiT, we do not support number statement with multiple origin functions.

The target code of the Person-Login model translated by BFiT is shown in appendix.

5 Evaluation and Future work

Despite the fact that BFiT is a prototype solution, it is still general enough to translate many BLOG models to Figaro correctly. We demonstrate the experiment result for 7 different BLOG models in Table 1, where the lines of code of both the original model in BLOG and the translated program in Figaro are presented.

From the empirical result, comparing with the original model in BLOG, the length of the program produced by BFiT has a bounded growth. This observation can be further converted to the following theorem.

Theorem 1. *BFiT always produces a target code in Figaro from a BLOG model with a constant blowup factor in program size.*

The proof directly follows the details of the translation approach for every kind of statement in BLOG.

Additionally, note that, in BFiT, we have carefully designed data structures in a general-purpose programming language (i.e., Scala) to maintain all the necessary information for an inference algorithm (i.e., all the objects and the memoized values for each random function). Our *dynamic memoization* technique in BFiT can be extended to a general compiler that directly produces a computation-specific program for fast inference in a canonical general-purpose programming language.

References

[1] N. Arora, S. Russell, P. Kidwell, and E. B. Sudderth. Global seismic monitoring as probabilistic inference. In *Advances in Neural Information Processing Systems*, pages 73–81, 2010.

[2] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *UAI*, pages 220–229, 2008.

[3] D. McAllester, B. Milch, and N. D. Goodman. Random-world semantics and syntactic independence for expressive languages. 2008.

[4] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[5] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, page 137, 2009.

[6] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

Appendix

The target code of the Person-Login model translated by BFiT is shown in Figure 6. For conciseness, we only show the code describing the model and ignore the part related to the inference algorithm.

```
var _MEMO_Honest = Map[(Person),Element[Boolean]]();
def B_Honest(x:Person):Element[Boolean]={
  val _IV_tup = (x);
  if (_MEMO_Honest.contains(_IV_tup)) return _MEMO_Honest(_IV_tup);
  val _IV_ret = Flip(0.9);
  _MEMO_Honest += _IV_tup -> _IV_0;
  return _IV_ret;
}; // denote the random function Honest() in BLOG

class Person(__name:Symbol) {val _name=__name;}; // class for type Person
val B_N_Person = Poisson(5); // number variable of Person
val B_AI_Person = // list that store all the objects of Person
  MakeList(B_N_Person,()=>Select(1.0->new Person('Person_#)));

class Login(__name:Symbol,__ORIGIN_Owner:Person){
  val _name=__name;
  val Owner =__ORIGIN_Owner; // field for origin function
}; // class for type Login

class B_G_AI_Login(_Owner:Person){
  val x = _Owner;
  val _n = If(B_Honest(x),Constant(1),Geometric(0.8));
  val _L = MakeList(_n,Select(1.0->new Login('Login_#,x)));
};
def _F_group_Login(_L_0:List[Person]):List[B_G_AI_Login]={
  var ret = new ListBuffer[B_G_AI_Login];
  for(_l_0<-_L_0) ret+=new B_G_AI_Login(_l_0);
  ret.toList
};
class B_AI_C_Login{
  lazy val _A = Apply(B_AI_Person,_F_group_Login);
  lazy val _total = Chain(_A,(_A:List[B_G_AI_Login])=>{
   val _B = Inject(_A.map(_._L):_*);
    Apply(_B,(_B:List[List[Login]])=>_B.flatten)
  });
}; // data structures for object generation

lazy val B_AI_Login = // store all the logins
  Chain(Select(1.0->new B_AI_C_Login),(b:B_AI_C_Login)=>b._total);
lazy val B_N_Login = // number variable of Login
  Apply(B_AI_Login,(_L:List[Login])=>_L.length);

lazy val B_sample = // a randomly sampled login
  Chain(B_AI_Login,(lst:List[Login])=>library.atomic.discrete.Uniform(lst:_*));
lazy val _QUERY_1 = Apply(Chain(B_sample,B_Honest), (x:Login)=>x.Owner); // query
```

Figure 6: translated target code for the Person-Login model