SWIFT: Compiled Inference for Probabilistic Programs

Lei Li
Yi Wu
Stuart J. Russell

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-12
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-12.html

March 27, 2015

Swift: Compiled Inference for Probabilistic Programs

Lei Li
Baidu Research
lilei22@baidu.com

Yi Wu
EECS Department
UC Berkeley
jxwuyi@cs.berkeley.edu

Stuart Russell
EECS Department
UC Berkeley
russell@cs.berkeley.edu

Abstract

One long-term goal for research on probabilistic programming languages (PPLs) is efficient inference using a single, generic inference engine. Many current inference engines are, however, *interpreters* for the given PP, leading to substantial overhead and poor performance. This paper describes a PPL compiler, Swift, that generates model-specific and inference-algorithm-specific target code from a given PP in the BLOG language, in much the same way that a Prolog compiler generates code that performs backward-chaining logical inference for a specific Prolog program. This approach eliminates a great deal of interpretation overhead. We evaluate the performance of Swift and existing systems such as BLOG, BUGS, Church, Stan, and infer.net on several benchmark problems. In our experiments, Swift's compiled code runs 100x faster than the original BLOG engine, and much faster than other systems on the same models.

1 Introduction

Probabilistic programming languages (PPLs) are a very promising approach for solving a long-standing problem in AI, machine learning, and statistics: the provision of an expressive, general-purpose modeling language capable of handling uncertainty, combined with a general-purpose inference engine able to handle any model the user might construct. The user (and, a fortiori, the brain) should not be required to carry out machine learning research and implement new algorithms for each problem that comes along.

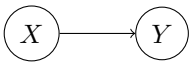
PPLs include BUGS [13] (and its variant JAGS [25]), BLOG [18], Church [6] (and its successor Venture [14]), Figaro [24], Markov logic networks (MLNs) [26], and Stan [29]. There have been a number of successes using PPLs for real applications such as entity resolution [28], citation matching [22], relation extraction [31], seismic monitoring [2], and decoding CAPTCHAs [15].

There are two main kinds of semantics for PPLs [16]: 1. possible-world semantics, which defines probability measures over complete assignments to sets of random variables; 2. random evaluation semantics, which defines probabilities over stochastic execution traces of a probabilistic program. The former kind includes BUGS, BLOG, MLN, and Stan. The latter includes Church and Figaro. A related property of a PPL is declarative versus imperative. The first group of languages are declarative, while the other are imperative. Not surprisingly the second group are all embedded languages, inheriting useful capabilities from their host languages (Lisp for Church and Scala for Figaro). There are other probabilistic systems that provide general probabilistic inference capabilities, such as Infer.NET [21] and FACTORIE [17]. Both define dependencies using factor graphs and perform inference over factors. AutoBayes [5] is a program synthesize system for a parameterized description of statistical models. Internally it uses Bayesian networks to describe dependencies. It can automatically generate optimized code to perform clustering analysis (through EM-algorithm) and numerical optimization. Our main focus in this paper is on compiling declarative PPLs with possible-world semantics—specifically, the BLOG language—although the approach is applicable to all other PPLs.

Inference for probabilistic programs (PPs) is very challenging, and the difficulty increases as the language grows more expressive. Real-world applications may require discrete and continuous variables, vector-valued variables, a rich library of distributions, the ability to describe relations and functions, and the ability to express *open-universe* models that allow for uncertainty about the existence and identity of objects. For example, the NETVISA seismic monitoring model [2] involves uncertainty about the number of seismic events that have occurred and the identity of the seismic event responsible for any particular observation is unknown. There are several recent algorithms proposed to solve the inference problem for open-universe probability models, including likelihood weighting (LW) [19], parental Metropolis-Hastings (MH) [20], a generalized form of Gibbs sampling algorithm (Gibbs) [1], and a form of ap-

proximate Bayesian computation (ABC) [15]. All of these algorithms operate by generating and manipulating data structures that correspond to possible worlds. While these algorithms could be improved and entirely new classes of algorithm are possible, our work in this paper is focused on achieving orders-of-magnitude improvement in the execution efficiency of a given algorithmic process.

This improvement is possible in many cases because most existing PPL inference engines (BUGS, BLOG, Church, Figaro, and JAGS) are *interpreters*. (The same is true even for Bayes net inference algorithms as they are commonly described and implemented.) What this means is that the PP—which expresses a probability model—exists within the inference engine in the form of a data structure (often after much internal preprocessing); a generic, model-independent inference algorithm consults this data structure to work out what calculation to do next. Because the model is fixed during any given run of the inference engine, the interpreter style results in a great deal of unnecessary work at run time answering questions—such as finding dependencies for a given variable, finding the procedure for sampling a given variable's conditional distribution, and so on—whose answers are already known at compile time. Consider the following simple example of a two-variable Bayes net, where the task is to perform a Gibbs step on Y:



Here are the steps taken by a typical inference algorithm: 1. It looks up the parent dependency record for Y, finding only X; 2. It looks up the child dependency record for Y, finding no children, and looks up their parents (none); 3. It looks up values of the parents (X), children (none) and children's parents (none) from the current possible world; 4. It obtains the conditional distribution for Y given its parents by substituting X's value in Y's CPD record, as well as the conditional distributions of the children (none); 5. It examines the types of all these conditional distributions to determine a procedure for sampling from their product; 6. It calls the procedure with appropriate inputs to generate a value for Y; 7. It stores the value of Y into the current possible world. Little of this work is necessary when the model is fixed. The compiler can generate *model-specific* inference code consisting of step 6, part of 3, and 7. Moreover, the "current possible world" can often be in the form of ordinary program variables with fixed machine addresses. The great majority of the CPU time in the compiled code should be spent inside the random number generator!

Our profiling results for the current BLOG inference engine show a significant portion (over 90%) of inference running time in BLOG is spent on steps other than sampling values from a conditional distribution, indicating that there is much to be gained from compilation. Motivated by these observations, we have developed a compiler, Swift, for PPs expressed in the BLOG language. (Reasons for choosing BLOG include its expressive power and its relatively small syntax compared to the embedded languages such as Church and Figaro.) For a given input PP and choice of inference algorithm, Swift generates C++ code that implements the inference algorithm in a form that is specialized to that particular PP. The C++ code is them compiled to machine code. For the example above, Swift generates two C++ subroutines, one for X and one for Y; each knows how to sample its own variable from precomputed, cached Gibbs distributions, indexed by the value of the other variable which is stored in a fixed, known machine address.

The contributions of the paper are as follows:

- We analyze the general software infrastructure required for compiling generative open-universe PPLs as well as specific techniques for particular inference algorithms.
- We describe an implemented, public-domain compiler, Swift, for the BLOG language; Swift handles likelihood-weighting, parental Metropolis-Hasting algorithm and Gibbs sampling for general BLOG models as well as particle filtering for temporal BLOG models.
- We report on experiments with a set of benchmark models as well as real applications, comparing Swift with the original BLOG engine and (where possible) with BUGS, Church, Stan, and Infer.NET. The results show that the code generated by Swift achieves roughly 100x speedup over the original BLOG engine and is significantly faster than other PPLs when executing comparable algorithms.

There are already existing efforts in compiling model descriptions into specialized execution code. AutoBayes [5], Infer.NET [21] and Stan [29] generate model-specific inference code (EM for AutoBayes, Message-Passing/Gibbs/Expectation-Propagation for Infer.NET, and Hamiltonian Monte Carlo for Stan) that doesn't use the model as a consulting data structure. The work of Huang et al [9] shares a similar compilation for MAP inference though their target is arithmetic circuits. All these PPLs are restricted to Bayesian networks, while our proposed Swift aims at compiling inference algorithms for open-universe probability models – its targeted BLOG is a fully expressive PPL with functions, relations, recursions and contingency.

2 Background

This sections describes basic elements of BLOG. We choose this language since it is based on possible-world semantics and is most expressive among existing languages of this kind.

2.1 BLOG: syntax and semantics

A BLOG program consists of a list of declaring statements for *type*, *constant symbols(i.e. objects)*, *fixed function*, *random function*, *observation* (i.e. data), and *query*. Random function declarations state the probabilistic dependencies among the random variables. In addition, the number of objects that belong to a *type* in a possible world is defined through number statements. The syntax is originally described in [18] and slightly evolved over years. An example of defining a Urn-Ball model in BLOG is:

```
1  type Ball; type Draw; type Color;
2  distinct Color Blue, Green; distinct Draw Draw[2];
3  #Ball ~ UniformInt(1,20);
4  random Color TrueColor(Ball b)
5    ~ Categorical({Blue -> 0.9, Green -> 0.1});
6  random Ball BallDrawn(Draw d)~UniformChoice({b for Ball b});
7  random Color ObsColor(Draw d) ~
8    case TrueColor(BallDrawn(d)) in {
9     Blue -> Categorical({Blue->0.9,Green->0.1}),
10    Green -> Categorical({Blue->0.1,Green->0.9}) };
11 obs ObsColor(Draw[0]) = Green;
12 obs ObsColor(Draw[1]) = Green;
13 query size({b for Ball b});
```

In this program, three types are declared: `Ball`, `Draw` and `Color`. There are two colors, `Green` and `Blue`, and two draws (i.e. two trials), which are defined in line 2. The number statement in line 3 declares that the total number (`\#Ball`) of ball is randomly distributed w.r.t. a uniform distribution over 1 to 20. The random function declaration on line 4 and 5 defines that, for each ball `b`, a random variable `TrueColor(b)` obeying a categorical distribution with 90% probability being `Blue`. Stated in line 6, Balls are drawn with replacement from the urn, the two draws being `BallDrawn(Draw[0])` and `BallDrawn(Draw[1])`.

The dependency defined in line 7 to 10 is called *context-specific*, since it uses `case-in` to specify noisy observations of `ObsColor(d)` based on the true color of the ball being drawn. In line 8, `TrueColor(BallDrawn(d))` is a *contingent variable* or a *switching variable*. Likewise, `BallDrawn(Draw[0])` and `BallDrawn(Draw[1])` are contingent on `#Ball`. A direct function application symbol such as `ObsColor(d)` for a concrete `Draw d` corresponds to a *basic random variable*. In contrast, a more complex symbol with multiple function compositions such as `TrueColor(BallDrawn(d))` corresponds derived variable. Since the value of `BallDrawn(d)` varies among possible worlds, the reference of `TrueColor(BallDrawn(d))` also varies.

Evidence is stated in line 11 and 12 with the `obs` keyword. Line 13 issues an query about the total number of balls in the urn.

Finally, note that this example models uncertainties not only in values of random variables, but also in the existence and identity of objects (due to number statement). We also call this kind of model, an *open-universe* probability model.

To sum up, BLOG is a possible-world-semantics based PPL that leverages the full expressive power of general contingent open-universe probabilistic models (OUPMs).

2.2 Generic Inference Algorithms

An expected answer to a query in OUPMs is the posterior distribution of the query expression given the evidence. One generic approach to answer such queries is through Monte Carlo sampling. Existing methods including rejection sampling, likelihood weighing algorithm(LW) [19], and Markov chain Monte Carlo algorithms such as parental Metropolis-Hasting algorithm (MH) [20] and Gibbs sampling [1]. For temporal models (i.e. models with `Timestep`), there are sequential Monte Carlo algorithm (SMC) such as particle filtering [7, 3] and Liu-West filter [12]. Both work for general models with arbitrary dependencies. Liu-West filter works better for models with both dynamics variables and continuous static parameters.

Before running into the generic inference algorithms, we introduction the notion of *supporting*: a variable x (in general can be any expression) is said to be *supported* in partial possible world PW if x's parents are instantiated and *supported* in PW. Essentially it means all ancestor variables of x are supported. Note that a variable x's ancestors can be different across possibles due to contingent dependencies. We also refer *parental* distribution of a variable to its defined conditional probability distribution in its BLOG program. We summarize these algorithms in Alg. (1,2,3,4). We use the following convention for these algorithms: $\mathcal{M}$ denotes an input BLOG program (or model), $\mathcal{E}$ its evidence, $\mathcal{Q}$ a query, and N denotes the number samples. For particle filtering, N is the number of particles and T time duration.

Existing systems execute these algorithms in a *interpreted* way. Take the LW algorithm (Alg. 1) for an example, the engine consults the input model $\mathcal{M}$ to obtain a parent variable with respect a possible world (line 5), to sample from its conditional probability distribution (line 6), and to calculate likelihood (line 8). These motivate our compilation techniques. After compiling into model-specific and inference-algorithm specific code, the inference code does not have to take the extra route to consult a model $\mathcal{M}$.

3 Compiled Inference

We are about to show snippets of machine-written code, because that is the easier way to both understand and author machine-writing code: abstract backwards from desired output (rather than simulate forwards from input). So, what optimizations are powerful and easy to automate? What optimizations are special to probabilistic programming? We will examine three model-specific optimizations, and six inference-specific optimizations.

Algorithm 1: Likelihood-Weighting (LW)

Input: $\mathcal{M}$, $\mathcal{E}$, $\mathcal{Q}$, N
Output: H: N samples and their associated weights

for $i \leftarrow 1$ *to* N **do**
  create empty possible world $PW_i \leftarrow \emptyset$, $w_i \leftarrow 1$;
  foreach *evidence* e_j *in* $\mathcal{E}$ **do**
    while e_j *is not* supported *in* PW_i **do**
      pick a *supported* variable x from ancestor set of e_j in PW_i;
      generate a value v from x's pdf in PW_i;
      $PW_i \leftarrow PW_i \cup \{x : v\}$;
    $w_i \leftarrow w_i \cdot pdf(e_j \mid PW_i)$;
  ensure $\mathcal{Q}$ is instantiated with a value v in PW_i;
  if $\mathcal{Q}$ *not instantiated* **then** generate a value v for $\mathcal{Q}$ from $\mathcal{Q}$'s parental distribution in PW_i;
  ;
  $H \leftarrow H \cup \{q : (v, w_i)\}$;

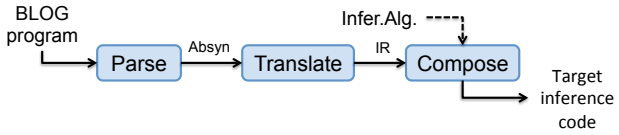


Figure 1: *Absyn*, a BLOG-specific abstract syntax tree, is the parsed model. *IR* decorates that with its meaning (resolve keywords, builtins, ...). *Compose* brings in the desired inference algorithm, producing the last intermediate representation. Final codegen targets C++.

3.1 Overview

Swift currently has three stages (Figure 1):

1. BLOG-models are parsed into *abstract syntax*.
2. Model-specific infrastructure choices (datastructures) are made and recorded, resulting in a new intermediate representation. See Section 3.2.
3. The selected inference algorithm (likelihood weighting, MCMC, ...) is combined with, and specialized for, the model. See Section 3.3).

The final intermediate representation is just an abstract syntax for (a subset of) C++: final codegen is trivial.[1] The key optimizations are in the two internal transformations, as well as in the optimizations we get for free by targeting another optimizing compiler.

3.2 Datastructure Optimizations

In BLOG, the basic random variables are types and functions. Consider a snippet of `UrnBall`:

[1]Our experiments use Microsoft C++ as the backend; in principle any other C++ compiler could be used. It would be interesting to experiment with Clang in particular, in order to leverage the advantages of LLVM.

Algorithm 2: Metropolis-Hasting sampling (MH)

Input: $\mathcal{M}$, $\mathcal{E}$, $\mathcal{Q}$, N
Output: H: a list of samples for $\mathcal{Q}$

$PW_0 \leftarrow$LW($\mathcal{M}$, $\mathcal{E}$, $\mathcal{Q}$, N);
for $i \leftarrow 1$ *to* N **do**
  $PW_i \leftarrow PW_{i-1}$;
  randomly pick a variable x from PW_i ;
  propose a value v from x's pdf in PW_i;
  update PW_i with $\{x : v\}$;
  using LW Alg. 1 to ensure $\mathcal{E}$ and $\mathcal{Q}$ supported in PW_i;
  $S \leftarrow$ the set of variables who differ in PW_i and PW_{i-1};
  $\alpha \leftarrow \frac{\prod_{x \in PW_i \wedge Par(x) \text{ differ in } PW_i, PW_{i-1}} pdf(x|PW_i, \mathcal{M})}{\prod_{y \in PW_{i-1} \wedge Par(y) \text{ differ in } PW_i, PW_{i-1}} pdf(x_j|PW_i)}$;
  $r \leftarrow \text{Uniform}(0, 1)$;
  if $r \geq \alpha$ **then** $PW_i \leftarrow PW_{i-1}$;
  ;
  $H \leftarrow H \cup \{\mathcal{Q} : PW_i(\mathcal{Q})\}$;

Algorithm 3: Gibbs sampling (Gibbs)

Input: $\mathcal{M}$, $\mathcal{E}$, $\mathcal{Q}$, N
Output: H: a list of samples for $\mathcal{Q}$

$PW_0 \leftarrow$LW($\mathcal{M}$, $\mathcal{E}$, $\mathcal{Q}$, N);
for $i \leftarrow 1$ *to* N **do**
  $PW_i \leftarrow PW_{i-1}$;
  randomly pick a variable x from PW_i ;
  if x *is Gibbs-doable* **then**
    $p \leftarrow pdf(x|PW_i, \mathcal{M})$;
    foreach *child* y *of* x *in* $\mathcal{M}$ *w.r.t.* PW_i **do**
      $p \leftarrow p \cdot pdf(y|PW_i, \mathcal{M})$;
    generate v from the distribution p;
    update PW_i with $\{x : v\}$;
  else
    use MH to propose and sample x;
  $H \leftarrow H \cup \{\mathcal{Q} : PW_i(\mathcal{Q})\}$;

```
type Ball; type Draw; distinct Draw d[2];
#Ball ~ UniformInt(1,20);
random Ball ball(Draw d) ~ UniformChoice({b for Ball b});
```

There are four basic random variables in this snippet: the set of balls, the set of draws, the first ball drawn (`ball(d[0])`), and the second ball drawn (`ball(d[1])`). One way to write one of the possible worlds is: $\langle$`Ball` $\rightarrow \{0\}$, `Draw` $\rightarrow \{$`d[0]`, `d[1]`$\}$, `ball(d[0])` $\rightarrow 0$, `ball(d[1])` $\rightarrow 0\rangle$. Since efficiency is our top concern, we describe below optimized machine representation of these variables.

3.2.1 Tables Over Maps

Generic maps are slow. Even hashmaps are slow; it can easily take hundreds of machine instructions to retrieve the

Algorithm 4: Particle-Filtering (PF)

Input: $\mathcal{M}, \mathcal{E}, \mathcal{Q}, T, N$

Output: $H^{1..T}$: generated samples for query at every time step

```
for i ∈ 1..N do initialize particle PW_i^0;
for t ← 1 to T do
    for i ← 1 to N do
        PW_i^t ← using LW to sample variables at
        Timestep t in M given the evidence E^t ;
        w_i ← pdf(E^t | PW_i^t, M) ;
    PW_{1..N}^t ← Multinomial({PW_{1..N}^t, w_{1..N}}) ;
    H^t ← {PW_{1..N}^t(Q^t)};
```

value of simple BLOG-terms such as `ball(d[0])`. A simple array, in contrast, permits retrieval in just a handful of instructions. Arrays are not always the best representation, of course. When we expect that answering the query will need just a few values from some function, then likely it is best to represent that function using some sparse representation (hashing, red-black trees, ...), despite the large constant overhead in access time. But if otherwise, then arrays/vectors/tables are simply too fast to be ignored.

Note that, for a compiler, it is straightforward to output different implementations for different parts of the model. For this example though, only dense representation makes sense. So, ideally, compiler-generated code for the example would resemble the handcrafted:

```
class World { public: int nBall; int ball[2];
  static World sample() { World ths;
    ths.nBall = Uniform::sample(1,20);
    ths.ball[0] = Uniform::sample(1,ths.nBall);
    ths.ball[1] = Uniform::sample(1,ths.nBall);
    return ths; }
  double probability() {
    assert(nBall >= 1 && nBall <= 20 &&
       ball[0] >= 1 && ball[0] <= nBall &&
       ball[1] >= 1 && ball[1] <= nBall);
    return 1. / (20. * nBall * nBall); } };
```

Swift does not output code that quite looks like that, but the performance is comparable. One complicating issue is that it is neither necessary nor desirable to fully construct worlds in order to answer queries. Rather, inference algorithms work with *partial* worlds. For that matter, inference algorithms typically need to associate various other kinds of meta-information to every variable of a model. So the template Swift actually follows is:

```
class ballT {
  int* value = new int[2]; int* mark = new int[2];
  /* children, evidence, prior, ... */ } ball;
```

Here `mark` is one way to implement a *partial* world; the value, when valid, of `ball(d[1])` is stored in `ball.value[1]`, and validity is whether `ball.mark[1]` is equal to the current mark. Additional meta-information, such as dependent variables, is simply added on as additional fields.

3.2.2 Lazy (Re-)Initialization

The `marks` are half of a standard technique for turning an eager computation into a lazy computation. The other piece of the puzzle is to force all access through procedures, so that the variable can be, as intended, initialized on demand. For example, the default *getter* initializes by sampling from the prior:

```
int ballT::get(int d) { if (mark[d] != currentMark) {
    value[d] = Uniform::sample(1,nBall.get());
    mark[d] = currentMark; }
  return value[d]; }
```

The marks only have to be reset whenever the current mark wraps around. There is an unexplored (by us) tradeoff here between the number of bits used to store the marks, and the frequency of having to reset all of them. (The minimum would be one bit per variable, and that would force resetting the marks every sample.) Presumably an `int` is too many bits; a mere byte-ful of `mark` would already put the frequency of resets at less than a half-percent.

Swift also generates a *setter* procedure in order to support directly observing the value of the variable (rather than just sampling from its prior). Many inference algorithms also need to calculate the prior probability of particular values; Swift likewise generates model-specific procedures for calculating *likelihoods*. Both of these follow the same template of checking the `marks`.

3.2.3 Open-Universe Types

The set of draws in this example is fixed ("closed"), so the representation of the `ball` function can just use simple arrays. The set of balls, however, is not. So, for the color of each ball (`color`), Swift uses a dynamic table (`vector`) rather than arrays:

```
class colorT{vector<int> value, mark; /* ... */} color;
```

For single-argument functions, `vector` gets the job done perfectly well.

With two or more arguments the design space becomes much larger, which we could investigate deeper in the future.[2] For now, Swift just uses nested dynamic tables, for example, `vector<vector<int>>` for a two argument function. That is not an ideal approach to multiple arguments, but it is easy to implement, and anyways is already an enormous improvement on typical implementations of general-purpose mapping datatypes.

[2]Morton-order indexing $(\ldots x_2x_1x_0, \ldots y_2y_1y_o) \rightarrow \ldots y_2x_2y_1x_1y_0x_0$ has the nice property that growing a dimension does not alter the index: the Morton index of (2,3) is always 14. In other words, growing a dimension does not force reshuffling the entire table, which — adding an object to an open type — is not an uncommon operation in open universes.

3.3 Inference Optimizations

Now we turn to six algorithm-specific optimizations. The first four are perhaps the most interesting, because they are rather specific to the nature of (first-order) probabilistic inference. The last two are normal instances of manually controlling memory for profit, which is theoretically quite mundane, but also quite practically significant.

3.3.1 Cyclic Dependencies

In BLOG, and many other languages, dependencies between functions may be stated cyclically, as long as each such recursion is 'well-behaved'. "Well-behaved" means that the induced dependencies on terms form an acyclic graph (in other words, every recursion has a base case, and every recursive case makes progress). For the most part, Swift does little to optimize for lack, or presence, of cycles, with some exceptions (deterministic and block dependencies). Rather than performance, actually, the most significant thing about cycles is implementation complexity. It is easy to fall into the habit of applying approaches that presuppose lack of cycles.

For example, the code above for lazily initializing a variable is not defensive. A user who gives a misbehaving model to the compiler will end up with inference code that also misbehaves (gets caught in an infinite loop). The BLOG-interpreter, in contrast, goes out of its way to check for 'obvious' infinite loops. Relative to that (friendly) interpreter, then, Swift optimizes by removing the runtime cycle checking.

3.3.2 Deterministic Dependencies

Sometimes there are deterministic dependencies among 'random' variables. For example, consider the following slightly modified snippet of `UrnBall`:

```
random Color obsColor(Draw d) = color(ball(d));
```

So we have made the observer perfect (the color of the draw is the color of the ball). In this case, if we naïvely apply likelihood weighting, the behavior will actually be that of rejection sampling (because the likelihood will either be 1, when we sample the matching color, or 0 otherwise). Note that, with a dependency this strong, setting evidence on `obsColor(d[0])` is equivalent to setting evidence on `color(ball(d[0]))`.

Generating code to propagate evidence is straightforward; Swift produces:

```
int obsColorT::set(int d, int v) { if (d==-1) return -1;
  mark[d] = currentMark;
  return (value[d] = color.set(ball.get(d),v));}
int colorT::set(int b, int v) { if (b==-1) return -1;
  mark[b] = currentMark;
  return (value[b] = v);}
```

A challenge for the future concerns *nearly*-deterministic dependencies. There, propagating evidence is not justified.

3.3.3 Block Proposals

Under various circumstances, particularly for MCMC, we need to generate one sample very much like another. If we are just changing one variable at a time, efficient implementation is straightforward. However, there are plenty of models where one cannot make any progress by just changing one variable. Particularly in open universes, adding and removing objects (*birth* and *death* moves) needs special treatment, because the likelihood of the proposed sample will be 0 unless all of the related variables are changed *en masse*.

For example, to change the number of balls (`#Ball`), almost every other variable in `UrnBall` needs to change. That is, which balls were drawn (tracked by `ball`) in the prior world cannot just be carried over, because the set of balls available to be drawn from has changed. (At a minimum there would be a bias that probably is not accounted for correctly.) Moreso the true colors of the balls (`color`) cannot just be carried over (because that will fail to delete the colors of any deleted balls, and will also fail to initialize the colors of any new balls). The only variables of `UrnBall` that do survive changing the set of balls are the observed colors of the draws, and the set of draws itself (which is constant).

Proposing to change many variables at once, efficiently, is a bit tricky. A 'correct' implementation would (a) copy the entire (partial) world, (b) make the changes, and (c) evaluate accept/reject. On reject, one would discard the copy, otherwise, the original. Performance-wise, the approach is incorrect, because copying an entire partial world in the inner loop is too expensive. To avoid copying during sampling, Swift preallocates enough storage for two partial worlds side by side (with some sharing of meta-information like children):

```
class colorT {
  vector<int> value, mark;
  vector<int> proposed, proposedMark;
  // ...
} color;
```

The *getters* and friends check `proposedMark`; if it matches the `currentMark` then `proposed` is taken instead of `value`. So the technique adds a little bit of cost on every access in order to avoid repeatedly copying a large structure only to change a small portion of it.

3.3.4 Markov Blankets

For inference in standard probabilistic graphical models, the Markov blankets can be easily precomputed. (A Markov blanket consists of the variable's parents, children, and children's parents.) However, in probabilistic programming languages, including BLOG, the dependencies between variables do not have a fixed structure across all possible worlds. So the Markov blankets are not, necessarily, constant.

For example, recall the form of `obsColor`'s dependency:

```
random Color obsColor(Draw d) ~
  case color(ball(d)) in { Blue -> ..., Green -> ... };
```

Suppose `ball(d)` changes from ball x to ball y. Then `obsColor(d)` becomes a child of `color(y)` rather than `color(x)`. In particular, the Markov blanket of `color` is not constant.

So we cannot, necessarily, precompute all the Markov blankets. Naturally, we could compute them all on the fly. While correct, doing so is dreadfully slow, because it completely fails to exploit the common case: the typical variable's blanket does not change from sample to sample. Instead, we have Swift generate code to incrementally propagate changes to Markov blankets. That is, we arrange the generated code so that it pays a cost proportional to the amount of change in the blankets from sample to sample. For the common case, this is a big win. In particular, if the BLOG-model happens to consist of nothing more than a Bayes Network, then all the blankets will be precomputed, and no computation will be spent updating them.

3.3.5 Pooling Particles (Custom Allocators)

There are two very important patterns of memory use: LIFO (stack), and FIFO (queue). These, among other patterns, are very important because they are extremely efficient ways to manage memory: one or two pointers and a pinch of arithmetic. Stack-allocation, especially, receives a lot of attention and special support.

Queues are also extremely natural, especially in strongly temporal settings. For our purposes, in implementing Sequential Monte Carlo inference algorithms, that is, particle filtering, we found it quite helpful to build in special support for queue-allocated data. For `K` particles in a `d`-order temporal model (meaning present transitions can depend on at most the `d` most recent states, which is a quantity Swift can and does precompute by analyzing the model), the Swift-generated code looks like:

```
Particle P[d+1][K];
Particle getParticle(int t,int i){return P[t%(d+1)][i];}
// ...
```

This is a queue implemented as a circular buffer. Of course, a bit of modular arithmetic is many hundreds of instructions faster than `malloc` and `free`.

3.3.6 (Avoiding) Duplication

When particle filtering, the resample step is a glaring opportunity to avoid unnecessary movement in memory. The resample step asks to create a set of uniformly weighted particles sampled according to the (nonuniform) weights they had previously. So, one expects to end up with duplicates of those particles that had the largest weights. However, rather than duplicate entire particles, we have Swift duplicate just pointers to the originals. So, for the benefit of the resample step, there is a parallel representation of particles using pointers instead:

```
Particle* ResampleP[d+1][K];
Particle* getResample(int t, int i)
  { return ResampleP[t%(d+1)][i]; }
// ...
```

4 Experimental results

In this section, we evaluate the performance of the Swift compiler for all the four algorithms mentioned above: Likelihood-Weighting (LW), parental Metropolis-Hasting, Gibbs Sampling and particle filtering (PF). For temporal modelsïijN̆ we adopt Liu-West filter [12] to estimate both dynamic variables and continuous static parameters. Swift uses C++ standard `<random>` library for random number generation and armadillo[27] package for matrix computation. The implementation details can be found in Appendix B.

The baseline systems are BUGS, BLOG interpreter, Church(WebChurch), Figaro, Infer.NET, Stan (CmdStan), all with the latest version. All experiments are run on a single machine with Intel quad-core 2.9GHz and 16G memory. Stan runs under Ubuntu 14.04, and the rest run on Windows 7. The system is configured with Java 8(jdk-1.8.25), Scala 2.11, Chrome browser 40. The detailed setup are in Appendix C. All PPLs are ensured to run in single-thread mode.

4.1 Benchmark models

We collect a set of benchmark models which exhibit various capabilities of a PPL (Table 1), including Burglary model (Burg), Hurricane model (Hurr), Tug-of-War (a simplified version of TrueSkill used in XBox [8]), Urn-Ball model with full open-universe uncertainty (Ball), 1 dimensional Gaussian mixture model (GMM), a hidden Markov model with four latent states (HMM). Experiments are run whenever a PPL is able to express the probability dependencies in the model. We measure the execution time of inference code using the same algorithm, excluding the compilation and data loading time. We include an additional comparison with Stan which has a unique inference algorithm (a variant of HMC).

We also include experiments for two models with real dataset: bird migration (Bird) and hand writing (using PPCA model). They are described separately. All the models can be found in Appendix A.

4.2 Likelihood-Weighting algorithm

LW is the most general algorithm for general contingent Open-Universe Bayesian networks despite of slow conver-

Table 1: Models used in experiments. D: discrete variables. R: continuous scalar or vector variables. CC: cyclic contingent dependency. OU: open-universe type. T: temporal models.

model feature	D	R	CC	OU	T
Burg	✓				
Hurr	✓		✓		
Tug-War	✓	✓			
Ball	✓			✓	
GMM	✓	✓			
HMM	✓				✓
Bird	✓				✓
PPCA		✓			

Table 2: Running time(s) for LW with 1 million samples.

model	Burg	Tug-War	Hurr	Ball
BLOG	8.42	79.6	19.8	188.2
Church	9.6	125.9	30.3	366.4
Figaro	14.6	453.5	24.7	333.0
Swift	**0.079**	**0.439**	**0.215**	**0.724**
speedup	107	181	92	260

gence. For this reason, we only test on four simpler models: Burglary, Tug-of-War, Hurricane, and Urn-Ball.

We compare the running time of generating 10^6 samples for PPLs supporting LW. The results are included in Table 2. Notice that Swift achieves over 100x speedup on average.

4.3 MCMC algorithms

There are two MCMC algorithms supported by the compiler, Parental MetropolisâĂŞHastings algorithm (MH) and Gibbs sampling. For MH, we compare against BLOG, Church, and Figaro, since the rest (BUGS and Infer.NET) do not provide MH. The running time for MH are shown in Table 3.

For Gibbs, we compare against BUGS and Infer.NET. We did not include Tug-War in this experiment since none of these systems is able to execute Tug-War using MH or Gibbs algorithm.

Table 3: Running time(s) for MCMC algorithms.

model	Burglary		Hurricane		Urn-Ball	
# Iter	10^5	10^6	10^5	10^6	10^5	10^6
Metropolis-Hastings						
BLOG	2.04	6.59	3.16	18.5	7.07	30.4
Church	1.35	12.7	2.56	25.2	28.7	379
Figaro	2.44	11.6	N/A	N/A	32.0	646
Swift	**0.014**	**0.15**	**0.026**	**0.242**	**0.069**	**0.7**
Gibbs Sampling						
BUGS	8.65	87.7	N/A	N/A	N/A	N/A
Infer.NET	0.175	1.50	N/A	N/A	N/A	N/A
Swift	**0.015**	**0.124**	**0.012**	**0.12**	**0.039**	**0.32**

Table 4: Running time(s) on GMM for Gibbs sampling.

#iter	10^4	10^5	10^6
BUGS	0.837	8.645	84.42
Infer.NET	0.823	7.803	77.83
Swift	**0.009**	**0.048**	**0.427**

Table 5: Running time (s) for PF. OOM: Out-of-memory.

Hidden Markov Model				
particle	10^3	10^4	10^5	10^6
BLOG	0.599	2.76	21.546	349.624
Figaro	0.799	2.102	11.527	135.155
Swift	**0.019**	**0.041**	**0.209**	**1.994**
Bird Migration Model				
particle	100	1000	5000	10^4
BLOG	1627.46	16902.2	OOM	OOM
Figaro	674.426	6811.95	30556.9	OOM
Swift	**11.6577**	**104.136**	**469.597**	**1038.187**

The running time on Burglary, Hurricane and Urn-Ball model are shown in Table 3.

Variables with finite support: Both Burglary and Hurricane models only contain discrete variables, it is fairly easy for PPL engine to compute the Gibbs density. Urn-Ball model has a number statement to determine the number of relevant variables in a possible world. The particular model has a finite support the number variable. Should it change to infinite (e.g. Poisson prior), Swift will automatically choose MH for the number of ball, and Gibbs for other variables.

BUGS and Infer.NET do not support Gibbs sampling on Hurricane Model and Urn-Ball model. On Burglary model, Swift achieves 10x speedup against Infer.NET and 707x speedup against BUGS as shown in Table 3.

Models with continuous variables: In order to apply Gibbs sampling to models with continuous variables, Swift require the random variables to have conjugate priors. Swift will automatically analyze the conjugacy. We compare the running time by BUGS, Infer.NET, and Swift on the GMM. The results are shown in Table 4. Swift achieves over 180x speedup comparing with BUGS and Infer.NET.

4.4 Particle filtering

For models with temporal dependencies, PF is a generic inference algorithm. We measure the running time of BLOG, Figaro and Swift on HMM and a real application, the bird migration model proposed in [4], with real-world data. Other PPLs are not evaluated since they do not support PF algorithm natively.

We ran the PF algorithm with various number of particles. The running time by different inference engines are shown in Table 5. Swift achieves over 50x speedup on average.

Bird Migration Problem: The bird migration problem is originally investigated in [4], which proposes a Hidden Markov Model to infer bird migration paths from a large database of observations. We apply our compiled particle filtering framework to the bird migration model in [4] using the dataset from the authors. In the dataset, the eastern continent of U.S.A is partitioned into 10x10 grids. There are roughly 10^6 birds totally observed in the dataset. For each grid, the total number of birds is observed over 60 days within 3 years. We aim to infer the number of birds migrating from each pair of grids between two consecutive days with observations.

To sum up, in the bird migration model, there are 60 states where each state contains 100 observed variables and 10^4 hidden variables. In order to handle continuous static parameters in the model, we apply Liu-West filter [12] here. We demonstrate the running time by Swift, the BLOG interpreter and Figaro with different number of particles in Table 5. When the number of particle increases, BLOG and Figaro do not produce an answer due to running out of the 16G memory.

In this real application, Swift achieves more than 100x speedup comparing with BLOG interpreter and more than 60x speedup against Figaro.

4.5 Comparing with Stan

Stan uses a different algorithm (HMC) and it generates compiled code as well. We compare both the inference effectiveness and efficiency of Stan and Swift on PPCA model, with real handwriting data.

Probabilistic principal component analysis (PPCA) is originally proposed in [30]. In PPCA, each observation $y_i \sim \mathcal{N}(Ax_i + \mu, \sigma^2 I)$, where A is a matrix with K columns, μ is the mean vector, and x_i is a coefficient vector associated with each data. All the entries of A, μ and x_i have independent Gaussian priors.

We use a subset of MNIST data set [11] for evaluation (corresponding to the digit "2"). The training and testing sets include 5958 and 1032 images respectively, each with 28x28 pixels. The pixel values are rescaled to the range $[0, 1]$. K is set to 10.

Note that Stan requires a tuning process before it can produce samples. We ran Stan multiple times with 0, 5 and 9 tuning steps respectively. We measure the perplexity of all the generated samples over the testing images from MNIST dataset. The perplexity with respect to the running time for Swift and Stan are shown in Figure 2 with the produced principal components visualized. We also ran Stan with 50 and 100 tuning steps (not shown in the figure), which took more than 3 days to finish 130 iterations (including tuning iterations). However, the perplexity of samples with 50 and 100 tuning iterations are almost the same as those with 9 tunning iterations. With 9 tuning steps (124s), Stan takes a total of 75.8s to generate 20 samples (0.26 sample per second). Swift takes 132s to generate 2 million samples (15k samples per second).

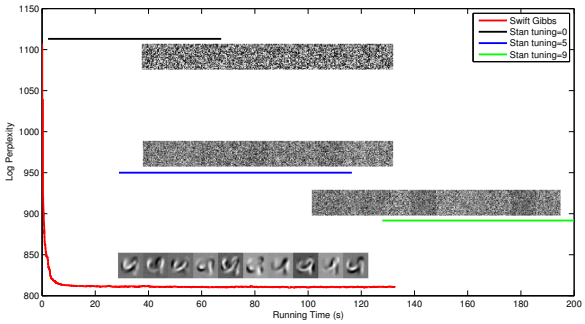


Figure 2: Log-perplexity w.r.t running time(s) on the PPCA model with visualized principal components. Swift converges faster, to a better result.

5 Conclusion

Limitations Two of Swift's limitations are worth briefly noting. While able to handle a significant subset of the BLOG language, certain gaps remain. In particular, Swift does not yet correctly compile recursively defined number statements. That prevents us from applying Swift to Probabilistic Context Free Grammars.

Swift does not, and should not be expected to, address any inherit limitations of the underlying inference algorithms. For example, the Open-Universe Gibbs algorithm does not support continuous-valued *switching* variables, meaning such cannot be used, in the interpreter or in Swift, in the guard of an `if` [1]. Noteworthiness is because verifying machine-written code for correctness is a much more daunting task than verifying either of general purpose interpreters or human-written model-specific code.

100x is Large Probabilistic programming languages (PPL) provide a general-purpose representation for modeling real world uncertainty. While modeling itself is often straightforward enough, making inference in these general frameworks work well is quite another matter. Most implementations thus far have been interpreted, or, when 'compiled', then the generated code and/or target language (e.g., Matlab) is still far above the nit and grit of real hardware. We conjectured, and demonstrated in the form of Swift, that getting those littlest of details right could and does make extremely large constant-factor improvements in speed. A hundredfold improvement is not unlike the difference between crawling and driving: too large to ignore.

References

[1] N. S. Arora, R. de Salvo Braz, E. B. Sudderth, and S. J. Russell. Gibbs sampling in open-universe stochastic languages. In P. Grï£¡nwald and P. Spirtes, editors, *UAI*, pages 30–39. AUAI Press, 2010.

[2] N. S. Arora, S. J. Russell, P. Kidwell, and E. B. Sudderth. Global seismic monitoring as probabilistic inference. In *NIPS*, pages 73–81, 2010.

[3] A. Doucet, N. De Freitas, and N. Gordon. *An introduction to sequential Monte Carlo methods*. Springer, 2001.

[4] M. Elmohamed, D. Kozen, and D. R. Sheldon. Collective inference on Markov models for modeling bird migration. In *Advances in Neural Information Processing Systems*, pages 1321–1328, 2007.

[5] B. Fischer and J. Schumann. AutoBayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13:483–508, 5 2003.

[6] 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.

[7] N. Gordon, D. Salmond, and A. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2):107–113, Apr 1993.

[8] R. Herbrich, T. Minka, and T. Graepel. Trueskill(tm): A bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, pages 569–576. MIT Press, January 2007.

[9] J. Huang, M. Chavira, and A. Darwiche. Solving map exactly by searching on compiled arithmetic circuits. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06*, pages 143–148, 2006.

[10] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] J. Liu and M. West. Combined parameter and state estimation in simulation-based filtering. In *Sequential Monte Carlo methods in practice*, pages 197–223. Springer, 2001.

[13] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. Winbugs – a bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, Oct. 2000.

[14] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*, Mar. 2014.

[15] V. K. Mansinghka, T. D. Kulkarni, Y. N. Perov, and J. B. Tenenbaum. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *NIPS*, pages 1520–1528, 2013.

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

[17] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, pages 1249–1257, 2009.

[18] 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.

[19] B. Milch, B. Marthi, D. Sontag, S. Russell, D. L. Ong, and A. Kolobov. Approximate inference for infinite contingent bayesian networks. In *Tenth International Workshop on Artificial Intelligence and Statistics, Barbados*, 2005.

[20] B. Milch and S. J. Russell. General-purpose MCMC inference over relational structures. In *UAI*. AUAI Press, 2006.

[21] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[22] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *Advances in neural information processing systems*, pages 1401–1408, 2002.

[23] V. Paxson. Flex, version 2.5. *URL http://www. gnu. org/software/flex*, 1990.

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

[25] M. Plummer et al. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, page 125. Vienna, 2003.

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

[27] C. Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

[28] P. Singla and P. Domingos. Entity resolution with Markov logic. In *ICDM*, pages 572–582, Dec 2006.

[29] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*, 2014.

[30] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 61:611–622, 1999.

[31] K. Yoshikawa, S. Riedel, M. Asahara, and Y. Matsumoto. Jointly identifying temporal relations with Markov logic. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 405–413, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

Supplemental Material for Swift: Compiled Inference for Probabilistic Programs

A Benchmark models

In this section, we briefly describe all the benchmark models and illustrate the corresponding BLOG program.

Burglary model (Burg): It is a hierarchical context-specific model containing 5 random boolean variables. It can be expanded to a Bayes net.

```
random Boolean Burglary
  ~ BooleanDistrib(0.001);
random Boolean Earthquake
  ~ BooleanDistrib(0.002);
random Boolean Alarm ~
  if Burglary then
    if Earthquake then BooleanDistrib(0.95)
    else  BooleanDistrib(0.94)
  else
    if Earthquake then BooleanDistrib(0.29)
    else BooleanDistrib(0.001);
random Boolean JohnCalls ~
  if Alarm then BooleanDistrib(0.9)
  else BooleanDistrib(0.05);
random Boolean MaryCalls ~
  if Alarm then BooleanDistrib(0.7)
  else BooleanDistrib(0.01);
obs JohnCalls = true;
obs MaryCalls = true;
query Burglary;
```

Tug-of-War (Tug-War): Two teams, each with two persons, play against each other. The winner teams are observed for each match. The query is to infer the power of each person. The model is a simplified version of TrueSkill used in XBox [8]. There is much determinism (e.g. summation) involved in the model. Only LW is tested on this model since plain MH and Gibbs do not work.

```
type Person;
type Match;
distinct Person Alice, Bob, Carl, Dan;
distinct Match M[3];
random Real strength(Person p)
  ~ Gaussian(10, 2);

fixed Person team1player1(Match m) =
  case m in
  {M[0]->Alice,M[1]->Alice,M[2]->Alice};
fixed Person team1player2(Match m) =
  case m in
  {M[0]->Bob,M[1]->Carl,M[2]->Dan};
fixed Person team2player1(Match m) =
  case m in
  {M[0]->Carl,M[1]->Bob,M[2]->Bob};
fixed Person team2player2(Match m) =
  case m in
  {M[0]->Dan,M[1]->Dan,M[2]->Carl};

random Boolean lazy(Person p, Match m)
  ~ BooleanDistrib(0.1);

random Real pulling_power(Person p, Match m)
  ~ if lazy(p, m) then strength(p) / 2.0
    else strength(p);

random Boolean team1win(Match m) ~
  if (pulling_power(team1player1(m), m)
      + pulling_power(team1player2(m), m)
    > pulling_power(team2player1(m), m)
      + pulling_power(team2player2(m), m) )
  then true
  else false;
obs team1win(M[0]) = true;
obs team1win(M[1]) = false;
obs team1win(M[2]) = false;
query strength(Alice) > strength(Bob);
```

Hurricane model (Hurr): The hurricane might attack two cities in a random order. The preparation level of the secondly attacked city is depending on the damage level of the first attacked city. This model contains *cyclic dependency* between variables which cannot be a concise Bayes net (though a full probability table could do). However, in any consistent possible world, the dependency among instantiated variables is acyclic.

```
type City;
type PrepLevel;
type DamageLevel;
distinct City A, B;
distinct PrepLevel High, Low;
distinct DamageLevel Severe, Mild;
random City First
  ~ Categorical({A->0.5,B->0.5});
random PrepLevel Prep(City c) ~
  if (First == c) then
    Categorical({High->0.5,Low->0.5})
  else
    case Damage(First) in {
      Severe ->
        Categorical({High->0.9,Low->0.1}),
      Mild ->
        Categorical({High->0.1,Low->0.9})
    };
random DamageLevel Damage(City c) ~
  case Prep(c) in {
    High ->
      Categorical({Severe->0.2,Mild->0.8}),
    Low ->
      Categorical({Severe->0.8,Mild->0.2})
  };
obs Damage(First) = Severe;
query Damage(A);
```

Urn-Ball model (Ball): The same as described in Section 2. It is an OUPM with unknown number of objects and their identity uncertainty.

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

distinct Color Blue, Green;
distinct Draw Draw[10];

#Ball ~ UniformInt(1,20);

random Color TrueColor(Ball b) ~
  Categorical({Blue -> 0.9, Green -> 0.1});

random Ball BallDrawn(Draw d) ~
  UniformChoice({b for Ball b});

random Color ObsColor(Draw d) ~
 case TrueColor(BallDrawn(d)) in {
  Blue ->
   Categorical({Blue -> 0.9, Green -> 0.1}),
  Green ->
   Categorical({Blue -> 0.1, Green -> 0.9})
 };

obs ObsColor(Draw[0]) = Green;
obs ObsColor(Draw[1]) = Green;
obs ObsColor(Draw[2]) = Green;
obs ObsColor(Draw[3]) = Green;
obs ObsColor(Draw[4]) = Green;
obs ObsColor(Draw[5]) = Green;
obs ObsColor(Draw[6]) = Green;
obs ObsColor(Draw[7]) = Green;
obs ObsColor(Draw[8]) = Green;
obs ObsColor(Draw[9]) = Blue;

query size({b for Ball b});
```

1-dimensional Gaussian mixture model (GMM): the model includes continuous variables with inverse Gamma prior on the variance. We generate 4 clusters with different mean and variance from the prior. We also generated 100 observations uniformly assigned to each of the clusters.

```
type Cluster; type Data;
distinct Cluster cluster[4];
distinct Data data[100];

random Real center(Cluster c)
  ~ Gaussian(0, 50);
random Real var(Cluster c)
  ~ InvGamma(1.0,1.0);
random Cluster Assign(Data d)
  ~ UniformChoice({c for Cluster c});
random Real Sample(Data d)
  ~ Gaussam(center(Assign(d)),
            var(Assign(d)));

query center(cluster[0]);
query center(cluster[1]);
query center(cluster[2]);
query center(cluster[3]);

obs Sample(data[0]) = ...;
obs Sample(data[1]) = ...;
...
obs Sample(data[99]) = ...;
```

Simple HMM Model: The HMM model contains 10 latent variables and 10 observed variables. Each latent variable may have 4 different values. This model is temporal, and its queries can be answered by PF.

```
type State;
distinct State A, C, G, T;

type Output;
distinct Output
  ResultA, ResultC, ResultG, ResultT;

random State S(Timestep t) ~
  if t == @0 then
    Categorical(
    {A -> 0.3, C -> 0.2, G -> 0.1, T -> 0.4})
  else case S(prev(t)) in {
    A -> Categorical(
    {A -> 0.1, C -> 0.3, G -> 0.3, T -> 0.3}),
    C -> Categorical(
    {A -> 0.3, C -> 0.1, G -> 0.3, T -> 0.3}),
    G -> Categorical(
    {A -> 0.3, C -> 0.3, G -> 0.1, T -> 0.3}),
    T -> Categorical(
    {A -> 0.3, C -> 0.3, G -> 0.3, T -> 0.1})
  };

random Output O(Timestep t) ~
  case S(t) in {
    A -> Categorical({
      ResultA -> 0.85, ResultC -> 0.05,
      ResultG -> 0.05, ResultT -> 0.05}),
    C -> Categorical({
      ResultA -> 0.05, ResultC -> 0.85,
      ResultG -> 0.05, ResultT -> 0.05}),
    G -> Categorical({
      ResultA -> 0.05, ResultC -> 0.05,
      ResultG -> 0.85, ResultT -> 0.05}),
    T -> Categorical({
      ResultA -> 0.05, ResultC -> 0.05,
      ResultG -> 0.05, ResultT -> 0.85})
  };

obs O(@1) = ResultA;
obs O(@2) = ResultA;
obs O(@3) = ResultA;
obs O(@4) = ResultG;
obs O(@5) = ResultG;
obs O(@6) = ResultG;
obs O(@7) = ResultG;
obs O(@8) = ResultT;
obs O(@9) = ResultC;
obs O(@10) = ResultA;

query S(@1);
query S(@2);
query S(@3);
query S(@4);
query S(@5);
query S(@6);
query S(@7);
query S(@8);
```

```
query S(@9);
query S(@10);
```

PPCA model: The PPCA model has been described in the main paper. Here is the corresponding BLOG program.

```
type Datapoint; type Dimension;
distinct Basis B[10];
distinct Datapoint datapoint[5958];

fixed Integer dim = 784;
fixed Integer bas = 10;
fixed Real sigma1 = 1;
fixed Real sigma2 = 1;
fixed Real sigma3 = 1;
fixed Real sigma4 = 1;

random RealMatrix basis(Basis b)
  ~ MultivarGaussian(zeros(dim),
                     sigma1 * eye(dim));
random RealMatrix mu
  ~ MultivarGaussian(zeros(dim),
                     sigma2 * eye(dim));

random RealMatrix x(Datapoint d)
  ~ MultivarGaussian(zeros(bas),
                     sigma3 * eye(bas));
random RealMatrix y(Datapoint d)
  ~ MultivarGaussian(
    hstack(EXPAND(B,0,9)) * x(d) + mu,
    sigma4 * eye(dim));

obs mu = zeros(dim);

obs y(datapoint[0]) = [];
obs y(datapoint[1]) = [];
...
obs y(datapoint[5957]) = [];

query basis(B[0]);
query basis(B[1]);
query basis(B[2]);
query basis(B[3]);
```

Bird Migration model: Here is the BLOG program for the bird migration problem. We do not show the `query` statements and `obs` statements for conciseness.

```
// defining the locations
type Location;
distinct Location l[100];
// parameters
random Real beta1 ~ UniformReal(3, 13);
random Real beta2 ~ UniformReal(3, 13);
random Real beta3 ~ UniformReal(3, 13);
random Real beta4 ~ UniformReal(3, 13);
// features
fixed RealMatrix F1(Location src) =
  loadRealMatrix("F1.txt", toInt(src));
fixed RealMatrix F2(Location src) =
  loadRealMatrix("F2.txt", toInt(src));
fixed RealMatrix F3(Location src, Timestep t)
  = loadRealMatrix("F3.txt",
        toInt(src) + toInt(t) * 100);
fixed RealMatrix F4(Location src) =
  loadRealMatrix("F4.txt", toInt(src));

// flow probabilities
random RealMatrix probs
  (Location src, Timestep t)
  ~ exp(beta1 * F1(src) + beta2 * F2(src)
      + beta3 * F3(src,t) + beta4 * F4(src));

// initial value for the birds
fixed Integer initial_value(Location loc) =
  if loc == l[0] then 1000000
  else 1;

// number of birds at location loc
//                    and timestep t
random Integer birds
  (Location loc, Timestep t) ~
  if t % 20 == @0 then initial_value(loc)
  else toInt(sum(
  { inflow(src, loc, prev(t))
            for Location src }));

// the vector of outflow from source(src)
//            to all other locations
random Integer[] outflow_vector
  (Location src, Timestep t) ~
  Multinomial(birds(src, t),
              transpose(probs(src, t)));

// inflow from source(src) to destination(dst)
random Integer inflow
  (Location src, Location dst, Timestep t) ~
  outflow_vector(src,t)[toInt(dst)];

// Noisy Observations
random Integer NoisyObs
  (Location loc, Timestep t) ~
  if birds(loc, t) == 0 then Poisson(0.01)
  else Poisson(birds(loc, t));
```

B Implementation

The compiler Swift is implemented in C++. We use C++ standard `<random>` library for random number generation. We manually write the cpd functions for all the supported random distributions using C++ standard `<cmath>` library. For matrix computation, we use armadillo[27] package.

Swift consists of 6 components: parser, model rewriter, semantics checker, IR analyzer, translator and printer. Each component is implemented as a single C++ class.

The parser uses YACC[10] and FLEX[23], which take in the model and produce the abstract syntax tree.

The model rewriter will expand the macros in the model to generate a full BLOG program.

The semantics checker takes in the abstract syntax tree and produce the intermediate representation. During this process, we will also rewrite IR in some cases for optimization purpose. For example, when given the following BLOG statement

```
random Ball draw~UniformChoice({b for Ball b});
```

the rewriter will rewrite it to

```
random Integer draw~UniformInt(0, #Ball - 1);
```

In the later statement, we do not need to explicitly construct a list containing all the balls.

The IR analyzer takes in IR and analyze the required information by the user selected algorithm. For example, the analyzer will compute the order of the Markov chain when Particle filtering algorithm is selected; for Gibbs sampling algorithm, the analyzer will do conjugacy analysis.

We implement a translator for each of the supported algorithms. The translators have lots of functions in common. A translator convert the IR to a simplified syntax tree targeting C++, which will be later printed to a well formated C++ program via printer.

We also implement lots of build in functions and data structures. For example, the resample step in particle filtering algorithm is manually implemented as a library function for efficiency; the multidimensional dynamic table data structure is implemented using template meta-programming and included in the built in library.

Our implementation is compatible with both g++ 4.8.1 and Visual Studio 2013.

C Details of experimental setup

The baseline PPLs for comparison include BUGS, BLOG (version 0.9.1)[3], Figaro (version 3.0.0), Church, Infer.NET (version 2.6)[4] and Stan. For BUGS, we use WinBUGS 1.4.3[5]. For Church, we use the latest version of WebChurch[6]. For Stan, we use CmdStan 2.6.0[7].

The experimental machine is equipped with Intel Core i7-3520 Quad-Core 2.90GHz and 16G memory. It is configured with Java 8 (jdk-1.8.25, for BLOG), Scala 2.11 (for Figaro), Visual Studio 2013 with default settings (for Swift and Infer.NET), Chrome browser Version 40.0.2214.115 (for WebChurch). CmdStan runs under Ubuntu 14.04 on the same machine. All other PPLs run under Windows 7.

D Additional results

D.1 Stan versus Swift on GMM

Besides the PPCA model with MNIST dataset, we also performed the experiment on the same GMM model with the same data as above. In order to measure accuracy, we generate another 100 samples from the ground truth as testing data. We compute the perplexity of the samples over the testing data.

Note that Stan requires a tuning process before it can produce samples. We run Stan for multiple times with different number of tuning steps. We ran Stan with 0, 4, 10, 40 and 100 tuning steps. The perplexity of the samples by Stan and Swift with respect to the running time are shown in Figure 3.

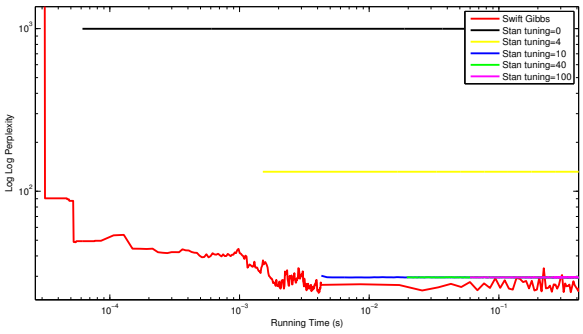


Figure 3: log-log-perplexity w.r.t running time(s) on the GMM.

[3] http://bayesianlogic.github.io

[4] http://research.microsoft.com/en-us/um/cambridge/projects/infernet/

[5] http://www.mrc-bsu.cam.ac.uk/software/bugs/the-bugs-project-winbugs/

[6] https://github.com/probmods/webchurch

[7] http://mc-stan.org/cmdstan.html