Swift: Compiled Inference for Probabilistic Programs

Yi Wu
UC Berkeley
jxwuyi@gmail.com

Lei Li
Baidu Research
lilei22@baidu.com

Stuart Russell
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 incur significant interpretation overhead . 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 with highly optimized data structures. We evaluate the performance of Swift and existing systems such as BLOG, BUGS, Church, Stan, and Infer.NET on several benchmark problems. In our experiment, the compiled code runs 100x faster than the benchmark systems.

1 Introduction

Probabilistic programming languages (PPLs) aim to combine sufficient expressive power for writing real-world probability models with efficient, general-purpose inference algorithms that can answer arbitrary queries with respect to those models.

PPLs have been applied to real-world tasks such as relation extraction [22], seismic monitoring [2], and decoding CAPTCHAs [7]. Well-known PPLs include IBAL [14], BUGS [5], JAGS [16], BLOG [9], Church [3], Venture [6], Figaro [15], Markov logic networks (MLNs) [17], FACTORIE [8], Infer.NET [12], Augur [21], and Stan [19]. Of these, BLOG, Church (also its variant Venture), and Figaro are Turing-equivalent in their expressive power; the other languages assume a known, finite number of random variables, often with a fixed dependency structure.

Generic inference methods for PPLs are mostly based on Monte Carlo approximation, including likelihood weighting (LW) [10], parental Metropolis–Hastings (MH) [11], a generalized form of Gibbs sampling (Gibbs) [1], Hamiltonian Monte Carlo (HMC) [13], and rejection sampling in the style of approximate Bayesian computation (ABC) [7].

By failing to take advantage of methods common in programming language implementation, existing PPLs give up two to three orders of magnitude in execution speed. The standard inference engines for IBAL, BUGS, JAGS, BLOG, Figaro, MLNs, and FACTORIE all run in *interpreted* mode, i.e., they execute generic inference code on a data structure that represents the probabilistic program; this incurs a very significant overhead. Church, Venture, Infer.NET, Augur, and Stan *compile* the probabilistic program into *program-specific* inference code. Whereas the three latter PPLs handle restricted input languages and achieve reasonable efficiency, the two former PPLs, which are Turing-equivalent, generate code that exhibits many opportunities for further optimization. Thus, our research focuses on compilation techniques required for efficient execution of inference algorithms for a Turing-equivalent PPL, in this case BLOG.

Inference for probabilistic programs (PPs) is very challenging, and the difficulty increases as the language grows more expressive. In models with *contingent dependencies*, the conditional-dependency relation changes in response to changes in values of random variables. Inference code that does not track the current dependency structure may perform sampling computations that are strictly irrelevant to the query answer. Furthermore, *open-universe* models allow for uncertainty about the existence (and thus the quantity) and identity of objects and their associated random variables. The

set of random variables defining the current inference state changes as inference proceeds, which creating new challenges for designing efficient data structures to maintain the current state.

We have developed a compiler, called Swift, able to efficiently accommodate open-universe models with contingent dependencies expressed in the BLOG language. It is notable that all the techniques in Swift are also extendable for other PPLs. Swift supports various inference algorithms including likelihood weighting (LW), Metropolis-Hastings algorithm (MH) and Gibbs sampling (Gibbs). For a given input PP and choice of inference algorithm, Swift generates highly optimized C++ code that implements the inference algorithm in a form that is specialized to that particular PP. The performance of Swift rivals that of hand-written, model-specific inference code, running at up to 60 million inference steps per second on a standard laptop.

2 Optimizations in Swift

We propose several techniques, including dynamic backchaining, reference counting and adaptive contingency updating to incrementally maintain the set of variables contributing to our computation target at run-time with a negligible overhead. Furthermore, we also carefully design data structures to reduce dynamic memory allocations as much as possible in the target code via lightweight memoization and efficient proposal manipulation.

Due to space limitation, please refer to the appendix for example compiled codes.

2.1 Optimizations for likelihood weighting algorithm

Dynamic backchaining: Dynamic backchaining (DB) aims to incrementally construct a minimal set of variables relevant to the query, namely a minimal self-supported possible world[1], at run-time. Our approach is based on the idea of lazy initialization: Swift instantiate a random variable and generates a random value from its declaration statement only when its value is required. By starting from evidence and queries, only the minimal number of random variables will be reached – all those relevant variables constitute a minimum self-supported partial world.

Lightweight memoization: In BLOG, every random variable is memoized. Swift analyzes the input PP and allocates static memory for memoization in the compiled code. For open-universe models where the number of random variables are unknown, the *dynamic table* data structure is used to reduce the amount of dynamic memory allocations. The dynamic table only allocates new memory when the number of variables in the new possible world increases during resampling; otherwise the pre-allocated memory will be reused.

On contrary to BLOG and Church, which maintains a string-based "name" for each random variable and uses a generic database (i.e., a hashmap) for storage, Swift adopts an integer to represent a particular variable and utilizes arrays for storage. In this framework, retrieving value for a random variable becomes direct indexing in a dense array, which eliminates a great deal of constant factor.

2.2 Optimizations for MCMC algorithms

Efficient proposal manipulation: In each iteration of MH or Gibbs, a proposal distribution is constructed for one random variable in focus. Although only one random variable's value is sampled, the step might introduce structural changes and instantiate/uninstantiate arbitrary number of random variables (i.e. when resampling a number variable)

The traditional safe but slow way in many existing PPL engines is to copy-construct a proposed possible world (PW), and the MCMC algorithm randomly decides to accept the proposed PW or to retain the original PW (Gibbs always accepts the new PW). Unfortunately, this suffers from tremendous memory allocation overhead. By contrast, Swift extends the lightweight memoization framework to support efficient proposal manipulation. Every random variable in the PP will have an extra cache, which is specially designed for storing the newly proposed value. When proposing a new possible world encountering new variables, the proposed value for the random variables not

[1] Intuitively, a possible world (PW) is an assignment of values to the random variables defined by the BLOG model. A detailed definition for possible world can be found in [9].

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

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

existing in the current possible world will be cached. When the proposed world gets accepted, we can incrementally achieve the current possible world by only retrieving those cached values.

Reference counting: During a resampling step, random variables might become obsolete in a proposed PW. Swift exploits an extra counter to random variables in the compiled code to track how many other random variables are directly depending on this random variable at run-time. After a resampling step, if a random variable has no references, we just remove it from the current possible world. The references of random variables are also incrementally maintained in Swift with a very lightweight overhead.

Adaptive contingency updating: In MH and Gibbs, when resampling a random variable, we need to compute its Markov blanket for efficient proposal generation. Since it is straightforward to find an random variable's parents in a given PP, the principal goal of adaptive contingency updating is to efficiently maintain the direct children of each random variable at run-time.

The dependency structure changes only happen when resample a *switching variable*[10]. Therefore, we only need to update the dependencies when a switching variable varies. All the switching variables in the input PP will be automatically analyzed by Swift at compile time. For each random variable, Swift maintains two sets. One is for all its children, namely all the variables directly depending on it. The other is for the variables that contingent on the current variable if it is a switching variable. We call the first set the children set and the latter the contingency set. When resampling a variable with a non-empty contingency set, the dependencies of all the variables in its contingency set will be updated.

3 Experimental results

We evaluate the performance of the Swift compiler for Likelihood-Weighting (LW), Metropolis-Hasting algorithm (MH) and Gibbs Sampling (Gibbs). Swift uses C++ standard `<random>` library for random number generation and armadillo[18] package for matrix computation.

The baseline systems are BUGS, BLOG (0.9.1), Church[2], Figaro (version 3.3.0), Infer.NET, Stan (CmdStan 2.7). All experiments are run on a standard laptop under single-thread mode.

3.1 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 (Tug-War, a simplified version of TrueSkill used in XBox [4]), Urn-Ball model with full open-universe uncertainty (Ball), 1 dimensional Gaussian mixture model (GMM, 100 data points and 4 clusters). 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) on a real dataset, MNIST (using PPCA model).

3.2 Likelihood-Weighting algorithm

LW is the most general algorithm for general Open-Universe probabilistic models despite of slow convergence. For this reason, we only test on four simpler models: Burglary, Tug-of-War, Hurricane, and Urn-Ball.

[2]webChurch: https://github.com/probmods/webchurch

model	Burg	Hurr	Ball
Metropolis-Hastings			
BLOG	6.74	18.5	30.4
Church	12.7	25.2	246.27
Figaro	10.58	N/A	59.59
Swift	**0.116**	**0.153**	**0.407**
Speedup	**58.1**	**121**	**74.7**
Gibbs Sampling			
BUGS	87.7	N/A	N/A
Infer.NET	1.50	N/A	N/A
Swift	**0.080**	**0.108**	**0.278**
Speedup	**18.75**	∞	∞

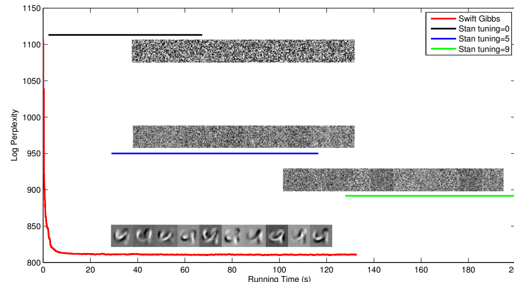
(a) Running time(s) for MCMC algorithms with 1 million samples.

#sample	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**
Speedup	**91.89**	**162.56**	**182.27**

(b) Running time(s) on GMM for Gibbs sampling.

model	Burg	Tug-War	Hurr	Ball
BLOG	8.42	79.6	11.85	146.432
Church	9.6	125.9	22.87	196.40
Figaro	15.83	453.5	24.7	148.64
Swift	**0.079**	**0.439**	**0.115**	**0.487**
speedup	**107**	**181**	**103**	**301**

(c) Running time(s) for LW with 1 million samples.



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

Figure 1: Experiment Results

We compare the running time of generating 10^6 samples for PPLs supporting LW. The results are included in Figure 1c. We notice that Swift achieves over 100x speedup on average.

3.3 MCMC algorithms

For MH, we compare against BLOG, Church, and Figaro. For Gibbs we compare with BUGS and Infer.NET. The running time on Burglary, Hurricane and Urn-Ball model are shown in Figure 1a.

BUGS and Infer.NET do not support Gibbs sampling on Hurricane Model and Urn-Ball model. On Burglary model, Swift achieves 18x speedup against Infer.NET and 1098x speedup against BUGS.

In order to apply Gibbs sampling to models with continuous variables, Swift requires the random variables to have conjugate priors. We compare the running time by BUGS, Infer.NET, and Swift on the GMM with 200 data points. The results are shown in Figure 1b. Swift achieves over 180x speedup comparing with BUGS and Infer.NET.

3.4 Comparing with Stan

Stan uses HMC for inference and also generates compiled code in C++. We compare both the inference effectiveness and efficiency of Stan and Swift (Swift uses Gibbs) on PPCA model, with a subset (all digits "2") containing 5958 and 1032 images for training and testing from MNIST dataset.

Probabilistic principal component analysis (PPCA) is originally proposed in [20]. In PPCA, each data $y_i \sim \mathcal{N}(Ax_i + \mu, \sigma^2 I)$, where A has K columns (we choose $K = 10$), and x_i is the coefficient for each data. All the entries of A, μ and x_i have independent Gaussian priors.

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 with respect to the running time of all the generated samples over the testing images from MNIST dataset in Figure 1d 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 tuning 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).

References

[1] N. S. Arora, R. de Salvo Braz, E. B. Sudderth, and S. J. Russell. Gibbs sampling in open-universe stochastic languages. In *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] 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.

[4] 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.

[5] 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.

[6] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] R. M. Neal. MCMC using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2, 2011.

[14] A. Pfeffer. IBAL: A probabilistic rational programming language. In *In Proc. 17th IJCAI*, pages 733–740. Morgan Kaufmann Publishers, 2001.

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

[16] 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.

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

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

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

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

[21] J.-B. Tristan, D. Huang, J. Tassarotti, A. C. Pocock, S. Green, and G. L. Steele. Augur: Data-parallel probabilistic modeling. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2600–2608. Curran Associates, Inc., 2014.

[22] 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.

Appendix

Let's consider the following fragment of BLOG program for the hurricane model[3].

```
type City; distinct City A, B;
type PrepLevel; distinct PrepLevel Low, High;
type DamageLevel; distinct DamageLevel Severe, Mild;

random City First ~ ...

random DamageLevel Damage(City c) ~ ...

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})
  };
```

We show a fragment of the compiled MH algorithm code for the random variable `Prep(·)` in the preceding model as follows. For description of the source code, please refer to the comments within the code.

```
// Basic data structure
//  defined for random var Prep()
class _Var_Prep{ public:
  int c; // denote the city argument
  int val; // stores the value of this random variable
  int is_active; // whether this var is relevant to the query
  int cache_val;// cache_val stores the proposal
  int is_cached;// memoization for the proposal
  int& getval();
  void sample();
  void active_edge();
}_mem_Prep[2];

// lightweight memoization
int& _Var_Prep::getval() {
  if(!is_active)
  { sample(); active_edge(); }
  return val;
}

// sample value for this random var
void _Var_Prep::sample() {
  if (_mem_First.getval()==c)
    val=Categorical0.gen();
  else switch (_mem_Damage[_mem_First.getval()].getval()) {
  case 0:
  val=Categorical1.gen(); break;
  case 1:
  val=Categorical2.gen(); break;
  }
}

// adaptive Markov Blanket Updating
void _Var_Prep::active_edge() {
  //First is the switching variable
  _mem_First.add_contig(this);
  _mem_First.add_child(this);
  if (_mem_First.getval()==c) {}
  else { _mem_Damage[
      _mem_First.getval()
    ].add_child(this); }
}
```

Figure 2: compiled C++ code for the Hurricane model

[3]The full hurricane model can be found at the website of BLOG, http://bayesianlogic.github.io/.