Swift: Compiled Inference for Probabilistic Programming Languages

Yi Wu
UC Berkeley
jxwuyi@gmail.com

Lei Li
Toutiao.com
lileicc@gmail.com

Stuart Russell
UC Berkeley
russell@cs.berkeley.edu

Rastislav Bodik
University of Washington
bodik@cs.washington.edu

Abstract

A probabilistic program defines a probability measure over its semantic structures. One common goal of probabilistic programming languages (PPLs) is to compute posterior probabilities for arbitrary models and queries, given observed evidence, using a generic inference engine. Most PPL inference engines—even the compiled ones—incur significant runtime interpretation overhead, especially for contingent and open-universe models. This paper describes Swift, a compiler for the BLOG PPL. Swift-generated code incorporates optimizations that eliminate interpretation overhead, maintain dynamic dependencies efficiently, and handle memory management for possible worlds of varying sizes. Experiments comparing Swift with other PPL engines on a variety of inference problems demonstrate speedups ranging from 12x to 326x.

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. One underlying motive is to relieve the user of the obligation to carry out machine learning research and implement new algorithms for each problem that comes along. Another is to support a wide range of cognitive functions in AI systems and to model those functions in humans.

General-purpose inference for PPLs is very challenging; they may include unbounded numbers of discrete and continuous variables, a rich library of distributions, and the ability to describe uncertainty over functions, relations, and the existence and identity of objects (so-called *open-universe* models). Existing PPL inference algorithms include likelihood weighting (LW) [Milch *et al.*, 2005b], parental Metropolis–Hastings (PMH) [Milch and Russell, 2006; Goodman *et al.*, 2008], generalized Gibbs sampling (Gibbs) [Arora *et al.*, 2010], generalized sequential Monte Carlo [Wood *et al.*, 2014], Hamiltonian Monte Carlo (HMC) [Stan Development Team, 2014], variational methods [Minka *et al.*, 2014; Kucukelbir *et al.*, 2015] and a form of approximate Bayesian computation [Mansinghka *et al.*, 2013]. While better algorithms are certainly possible, our focus in this paper is on achieving orders-of-magnitude improvement in the execution efficiency of a given algorithmic process.

A PPL system takes a probabilistic program (PP) specifying a probabilistic model as its input and performs inference to compute the posterior distribution of a *query* given some observed *evidence*. The inference process does not (in general) *execute* the PP, but instead executes the steps of an inference algorithm (e.g., Gibbs) guided by the dependency structures implicit in the PP. In many PPL systems the PP exists as an internal data structure consulted by the inference algorithm at each step [Pfeffer, 2001; Lunn *et al.*, 2000; Plummer, 2003; Milch *et al.*, 2005a; Pfeffer, 2009]. This process is in essence an *interpreter* for the PP, similar to early Prolog systems that interpreted the logic program. Particularly when running sampling algorithms that involve millions of repetitive steps, the overhead can be enormous. A natural solution is to produce model-specific compiled inference code, but, as we show in Sec. 2, existing compilers for general open-universe models [Wingate *et al.*, 2011; Yang *et al.*, 2014; Hur *et al.*, 2014; Chaganty *et al.*, 2013; Nori *et al.*, 2014] miss out on optimization opportunities and often produce inefficient inference code.

The paper analyzes the optimization opportunities for PPL compilers and describes the Swift compiler, which takes as input a BLOG program [Milch *et al.*, 2005a] and one of three inference algorithms (LW, PMH, Gibbs) and generates target code for answering queries. Swift includes three main contributions: (1) elimination of interpretative overhead by joint analysis of the model structure and inference algorithm; (2) a dynamic slice maintenance method (FIDS) for incremental computation of the current dependency structure as sampling proceeds; and (3) efficient runtime memory management for maintaining the current-possible-world data structure as the number of objects changes. Comparisons between Swift and other PPLs on a variety of models demonstrate speedups ranging from 12x to 326x, leading in some cases to performance comparable to that of hand-built model-specific code. To the extent possible, we also analyze the contributions of each optimization technique to the overall speedup.

Although Swift is developed for the BLOG language, the overall design and the choices of optimizations can be applied to other PPLs and may bring useful insights to similar AI

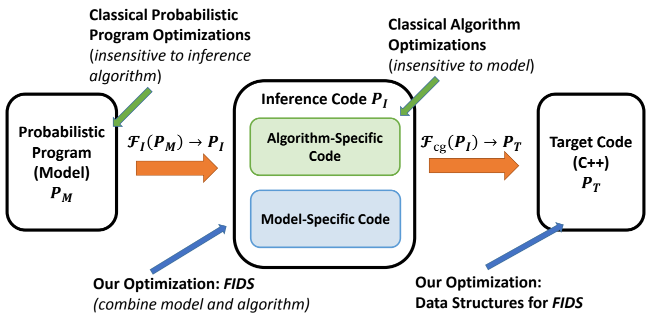
Classical Probabilistic Program Optimizations
(insensitive to inference algorithm)
Classical Algorithm Optimizations
(insensitive to model)
Probabilistic Program (Model) P_M
$\mathcal{F}_I(P_M) \rightarrow P_I$
Inference Code P_I
Algorithm-Specific Code
Model-Specific Code
$\mathcal{F}_{cg}(P_I) \rightarrow P_T$
Target Code (C++) P_T
Our Optimization: FIDS
(combine model and algorithm)
Our Optimization: Data Structures for FIDS

Figure 1: PPL compilers and optimization opportunities.

systems for real-world applications.

2 Existing PPL Compilation Approaches

In a general purpose programming language (e.g., C++), the compiler compiles exactly what the user writes (the program). By contrast, a PPL compiler essentially compiles the inference algorithm, which is written by the PPL developer, as applied to a PP. This means that different implementations of the same inference algorithm for the same PP result in completely different target code.

As shown in Fig. 1, a PPL compiler first produces an intermediate representation combining the inference algorithm (I) and the input model (P_M) as the *inference code* (P_I), and then compiles P_I to the *target code* (P_T). Swift focuses on optimizing the inference code P_I and the target code P_T given a fixed input model P_M.

Although there have been successful PPL compilers, these compilers are all designed for a restricted class of models with fixed dependency structures: see work by Tristan [2014] and the Stan Development Team [2014] for closed-world Bayes nets, Minka [2014] for factor graphs, and Kazemi and Poole [2016] for Markov logic networks.

Church [Goodman *et al.*, 2008] and its variants [Wingate *et al.*, 2011; Yang *et al.*, 2014; Ritchie *et al.*, 2016] provide a lightweight compilation framework for performing the Metropolis–Hastings algorithm (MH) over general open-universe probability models (OUPMs). However, these approaches (1) are based on an inefficient implementation of MH, which results in overhead in P_I, and (2) rely on inefficient data structures in the target code P_T.

For the first point, consider an MH iteration where we are proposing a new possible world w' from w accoding to the proposal $g(\cdot)$ by resampling random variable X from v to v'. The computation of acceptance ratio α follows

$$\alpha = \min\left(1, \frac{g(v' \rightarrow v)\,\Pr[w']}{g(v \rightarrow v')\,\Pr[w]}\right). \tag{1}$$

Since only X is resampled, it is sufficient to compute α using merely the Markov blanket of X, which leads to a much simplified formula for α from Eq.(1) by cancelling terms in $\Pr[w]$ and $\Pr[w']$. However, when applying MH for a contingent model, the Markov blanket of a random variable cannot be determined at compile time since the model dependencies vary during inference. This introduces tremendous runtime overhead for interpretive systems (e.g., BLOG, Figaro),

```
1  type Ball; type Draw; type Color; //3 types
2  distinct Color Blue, Green;  //two colors
3  distinct Draw D[2]; //two draws: D[0], D[1]
4  #Ball ~ UniformInt(1,20);//unknown # of balls
5  random Color color(Ball b) //color of a ball
6    ~ Categorical({Blue -> 0.9, Green -> 0.1});
7  random Ball drawn(Draw d)
8    ~ UniformChoice({b for Ball b});
9  obs color(drawn(D[0])) = Green; // drawn ball is green
10 query color(drawn(D[1]));  // query
```

Figure 2: The urn-ball model

which track all the dependencies at runtime even though typically only a tiny portion of the dependency structure may change per iteration. Church simply avoids keeping track of dependencies and uses Eq.(1), including a potentially huge overhead for redundant probability computations.

For the second point, in order to track the existence of the variables in an open-universe model, similar to BLOG, Church also maintains a complicated dynamic string-based hashing scheme in the target code, which again causes interpretation overhead.

Lastly, techniques proposed by Hur *et al.* [2014], Chaganty *et al.* [2013] and Nori *et al.* [2014] primarily focus on optimizing P_M by analyzing the static properties of the input PP, which are complementary to Swift.

3 Background

This paper focuses on the BLOG language, although our approach also applies to other PPLs with equivalent semantics [McAllester *et al.*, 2008; Wu *et al.*, 2014]. Other PPLs can be also converted to BLOG via static single assignment form (SSA form) transformation [Cytron *et al.*, 1989; Hur *et al.*, 2014].

3.1 The BLOG Language

The BLOG language [Milch *et al.*, 2005a] defines probability measures over first-order (relational) possible worlds; in this sense it is a probabilistic analogue of first-order logic. A BLOG program declares *types* for objects and defines distributions over their numbers, as well as defining distributions for the values of random functions applied to objects. *Observation statements* supply evidence, and a *query* statement specifies the posterior probability of interest. A random variable in BLOG corresponds to the application of a random function to specific objects in a possible world. BLOG naturally supports open universe probability models (OUPMs) and context-specific dependencies.

Fig. 2 demonstrates the open-universe urn-ball model. In this version the query asks for the color of the next random pick from an urn given the colors of balls drawn previously. Line 4 is a *number statement* stating that the *number variable* $\#Ball$, corresponding to the total number of balls, is uniformly distributed between 1 and 20. Lines 5–6 declare a random function $color(\cdot)$, applied to balls, picking `Blue` or `Green` with a biased probability. Lines 7–8 state that each draw chooses a ball at random from the urn, with replacement.

Fig. 3 describes another OUPM, the infinite Gaussian mixture model (∞-GMM). The model includes an unknown

```
1 type Cluster; type Data;
2 distinct Data D[20]; // 20 data points
3 #Cluster ~ Poisson(4); // number of clusters
4 random Real mu(Cluster c) ~ Gaussian(0,10); //cluster mean
5 random Cluster z(Data d)
6   ~ UniformChoice({c for Cluster c});
7 random Real x(Data d) ~ Gaussian(mu(z(d)), 1.0); // data
8 obs x(D[0]) = 0.1; // omit other data points
9 query size({c for Cluster c});
```

Figure 3: The infinite Gaussian mixture model

Algorithm 1: Metropolis–Hastings algorithm (MH)

```
Input: $\mathbf{M}$, $\mathbf{E}$, $\mathbf{Q}$, $N$; Output: samples $H$
initialize a possible world $w^{(0)}$ with $\mathbf{E}$ satisfied;
for $i \leftarrow 1$ to $N$ do
    randomly pick a variable $X$ from $w^{(i-1)}$ ;
    $w^{(i)} \leftarrow w^{(i-1)}$ and $v \leftarrow X(w^{(i-1)})$;
    propose a value $v'$ for $X$ via proposal $g$;
    $X(w^{(i)}) \leftarrow v'$ and ensure $w^{(i)}$ is self-supporting;
    $\alpha \leftarrow \min\left(1, \frac{g(v' \to v)\Pr[w^{(i)}]}{g(v \to v')\Pr[w^{(i-1)}]}\right)$ ;
    if $\mathrm{rand}(0,1) \geq \alpha$ then $w^{(i)} \leftarrow w^{(i-1)}$
    $H \leftarrow H + (\mathbf{Q}(w^{(i)}))$;
```

number of clusters as stated in line 3, which is to be inferred from the data.

3.2 Generic Inference Algorithms

All PPLs need to infer the posterior distribution of a query given the observed evidence. The great majority of PPLs use Monte Carlo sampling methods such as likelihood weighting (LW) and Metropolis–Hastings MCMC (MH). LW samples unobserved random variables sequentially in topological order, conditioned on evidence and sampled values earlier in the sequence; each complete sample is weighted by the likelihood of the evidence variables appearing in the sequence. The MH algorithm is summarized in Alg. 1. $\mathbf{M}$ denotes a BLOG model, $\mathbf{E}$ its evidence, $\mathbf{Q}$ a query, N the number of samples, w a possible world, $X(w)$ the value of random variable X in w, $\Pr[X(w)|w_{\text{-}X}]$ denotes the conditional probability of X in w, and $\Pr[w]$ denotes the likelihood of possible world w. In particular, when the proposal distribution g in Alg.1 samples a variable conditioned on its parents, the algorithm becomes the parental MH algorithm (PMH). Our discussion in the paper focuses on LW and PMH but our proposed solution applies to other Monte Carlo methods as well.

4 The Swift Compiler

Swift has the following design goals for handling open-universe probability models.

- Provide a general framework to automatically and efficiently (1) track the exact Markov blanket for each random variable and (2) maintain the minimum set of random variables necessary to evaluate the query and the evidence (the dynamic slice [Agrawal and Horgan, 1990], or equivalently the minimum self-supporting partial world [Milch *et al.*, 2005a]).
- Provide efficient memory management and fast data structures for Monte Carlo sampling algorithms to avoid interpretive overhead at runtime.

For the first goal, we propose a novel framework, FIDS (Framework for Incrementally updating Dynamic Slices), for generating the inference code (P_I in Fig. 1). For the second goal, we carefully choose data structures in the target C++ code (P_T).

For convenience, r.v. is short for random variable. We demonstrate examples of compiled code in the following discussion: the inference code (P_I) generated by FIDS in Sec. 4.1 is in pseudo-code while the target code (P_T) by Swift shown in Sec. 4.2 is in C++. Due to limited space, we show the detailed transformation rules only for the adaptive contingency updating (ACU) technique and omit the others.

4.1 Optimizations in FIDS

Our discussion in this section focuses on LW and PMH, although FIDS can be also applied to other algorithms. FIDS includes three optimizations: dynamic backchaining (DB), adaptive contingency updating (ACU), and reference counting (RC). DB is applied to both LW and PMH while ACU and RC are specific to PMH.

Dynamic backchaining

Dynamic backchaining (DB) [Milch *et al.*, 2005a] constructs a dynamic slice incrementally in each LW iteration and samples only those variables necessary at runtime. DB in Swift is an example of compiling lazy evaluation: in each iteration, DB backtracks from the evidence and query and samples an r.v. only when its value is required during inference.

For every r.v. declaration `random T` $X \sim C_X$`;` in the input PP, FIDS generates a *getter* function `get_X()`, which is used to (1) sample a value for X from its declaration C_X and (2) memoize the sampled value for later references in the current iteration. Whenever an r.v. is referred to during inference, its getter function will be evoked.

Compiling DB is the fundamental technique for Swift. The key insight is to replace dependency look-ups and method invocations from some internal PP data structure with direct machine address accessing and branching in the target executable file.

For example, the inference code for the getter function of $x(d)$ in the ∞-GMM model (Fig. 3) is shown below.

```
double get_x(Data d) {
  // some code memoizing the sampled value
  memoization;
  // if not sampled, sample a new value
  val = sample_gaussian(get_mu(get_z(d)),1);
  return val; }
```

`memoization` denotes some pseudo-code snippet for performing memoization. Since `get_z`$(\cdot)$ is called before `get_mu`$(\cdot)$, only those $mu(c)$ corresponding to non-empty clusters will be sampled. Notably, with the inference code above, no explicit dependency look-up is ever required at runtime to discover those non-empty clusters.

When sampling a number variable, we need to allocate memory for the associated variables. For example, in the urn-ball model (Fig. 2), we need to allocate memory for $color(b)$

after sampling $\#Ball$. The corresponding generated inference code is shown below.

```
int get_num_Ball() {
  // some code for memoization
  memoization;
  // sample a value
  val = sample_uniformInt(1,20);
  // some code allocating memory for color
  allocate_memory_color(val);
  return val; }
```

`allocate_memory_color(val)` denotes some pseudo-code segment for allocating `val` chunks of memory for the values of $color(b)$.

Reference Counting

Reference counting (RC) generalize the idea of DB to incrementally maintain the dynamic slice in PMH. RC is an efficient compilation strategy for the interpretive BLOG to dynamically maintain references to variables and exclude those without any references from the current possible world with minimal runtime overhead. RC is also similar to dynamic garbage collection in programming language community.

For an r.v. being tracked, say X, RC maintains a reference counter `cnt(X)` defined by `cnt(X)` $= |Ch_w(X)|$. $Ch_w(X)$ denote the children of X in the possible world w. When `cnt(X)` becomes zero, X is removed from w; when `cnt(X)` become positive, X is instantiated and added back to w. This procedure is performed recursively.

Tracking references to every r.v. might cause unnecessary overhead. For example, for classical Bayes nets, RC never excludes any variables. Hence, as a trade-off, Swift only counts references in open-universe models, particularly, to those variables associated with number variables (e.g., $color(b)$ in the urn-ball model).

Take the urn-ball model as an example. When resampling $drawn(d)$ and accepting the proposed value v, the generated code for accepting the proposal will be

```
void inc_cnt(X) {
  if (cnt(X) == 0) W.add(X);
  cnt(X) = cnt(X) + 1; }
void dec_cnt(X) {
  cnt(X) = cnt(X) - 1;
  if (cnt(X) == 0) W.remove(X); }
void accept_value_drawn(Draw d, Ball v) {
// code for updating dependencies omitted
  dec_cnt(color(val_drawn(d)));
  val_drawn(d) = v;
  inc_cnt(color(v)); }
```

The function `inc_cnt(X)` and `dec_cnt(X)` update the references to X. The function `accept_value_drawn()` is specialized to r.v. $drawn(d)$: Swift analyzes the input program and generates specialized codes for different variables.

Adaptive contingency updating

The goal of ACU is to incrementally maintain the Markov Blanket for every r.v. with minimal efforts.

C_X denotes the declaration of r.v. X in the input PP; $Par_w(X)$ denote the parents of X in the possible world w; $w[X \leftarrow v]$ denotes the new possible world derived from w by only changing the value of X to v. Note that deriving $w[X \leftarrow v]$ may require instantiating new variables not existing in w due to dependency changes.

Computing $Par_w(X)$ for X is straightforward by executing its generation process, C_X, within w, which is often inexpensive. Hence, the principal challenge for maintaining the Markov blanket is to efficiently maintain a *children set* `Ch(X)` for each r.v. X with the aim of keeping identical to the true set $Ch_w(X)$ for the current possible world w.

With a proposed possible world (PW) $w' = w[X \leftarrow v]$, it is easy to add all missing dependencies from a particular r.v. U. That is, for an r.v. U and every r.v. $V \in Par_{w'}(U)$, since U must be in $Ch_{w'}(V)$, we can keep `Ch(V)` up-to-date by adding U to `Ch(V)` if $U \notin$ `Ch(V)`. Therefore, the key step is tracking the set of variables, $\Delta(w[X \leftarrow v])$, which will have a set of parents in $w[X \leftarrow v]$ different from w.

$$\Delta(W[X \leftarrow v]) = \{Y : Par_w(Y) \neq Par_{w[X \leftarrow v]}(Y)\}$$

Precisely computing $\Delta(w[X \leftarrow v])$ is very expensive at runtime but computing an upper bound for $\Delta(w[X \leftarrow v])$ does not influence the correctness of the inference code. Therefore, we proceed to find an upper bound that holds for any value v. We call this over-approximation the *contingent set* $Cont_w(X)$.

Note that for every r.v. U with dependency that changes in $w(X \leftarrow v)$, X must be reached in the condition of a control statement on the execution path of C_U within w. This implies $\Delta(w[X \leftarrow v]) \subseteq Ch_w(X)$. One straightforward idea is set $Cont_w(X) = Ch_w(X)$. However, this leads to too much overhead. For example, $\Delta(\cdot)$ should be always empty for models with fixed dependencies.

Another approach is to first find out the set of *switching variables* S_X for every r.v. X, which is defined by

$$S_X = \{Y : \exists w, v \ Par_w(X) \neq Par_{w[Y \leftarrow v]}(X)\},$$

This brings an immediate advantage: S_X can be statically computed at compile time. However, computing S_X is NP-Complete[1].

Our solution is to derive the approximation $\widehat{S}_X$ by taking the union of free variables of if/case conditions, function arguments, and set expression conditions in C_X, and then set

$$Cont_w(X) = \{Y : Y \in Ch_w(X) \wedge X \in \widehat{S}_Y\}.$$

$Cont_w(X)$ is a function of the sampling variable X and the current PW w. Likewise, for every r.v. X, we maintain a runtime contingent set `Cont(X)` identical to the true set $Cont_w(X)$ under the current PW w. `Cont(X)` can be incrementally updated as well.

Back to the original focus of ACU, suppose we are adding new the dependencies in the new PW $w' = w[X \leftarrow v]$. There are three steps to accomplish the goal: (1) enumerating all the variables $U \in$ `Cont(X)`, (2) for all $V \in Par_{w'}(U)$, add U to `Ch(V)`, and (3) for all $V \in Par_{w'}(U) \cap \widehat{S}_U$, add U to `Cont(V)`. These steps can be also repeated in a similar way to remove the vanished dependencies.

Take the ∞-GMM model (Fig. 3) as an example , when resampling $z(d)$, we need to change the dependency of r.v. $x(d)$

[1]Since the declaration C_X may contain arbitrary boolean formulas, one can reduce the 3-SAT problem to computing S_X.

```
$\mathcal{F}_c(\mathbf{M} = \texttt{random}\ \mathbf{T}\ \mathbf{Id}([\mathbf{T}\ \mathbf{Id}\ ,]^*) \sim \mathbf{C}) =$
      void Id::add_to_Ch()
          { $\mathcal{F}_{cc}(\mathbf{C}, \mathbf{Id}, \{\})$ }
       void Id::accept_value(T v)
        { for(u in Cont(Id)) u.del_from_Ch();
          Id = v;
          for(u in Cont(Id)) u.add_to_Ch(); }

$\mathcal{F}_{cc}(\mathbf{C} = \mathbf{Exp}, X, seen) = \mathcal{F}_{cc}(\mathbf{Exp}, X, seen)$
$\mathcal{F}_{cc}(\mathbf{C} = \mathbf{Dist}(\mathbf{Exp}), X, seen) = \mathcal{F}_{cc}(\mathbf{Exp}, X, seen)$
$\mathcal{F}_{cc}(\mathbf{C} = \texttt{if}\ (\mathbf{Exp})\ \texttt{then}\ \mathbf{C}_1\ \texttt{else}\ \mathbf{C}_2, X, seen) =$
      $\mathcal{F}_{cc}(\mathbf{Exp}, X, seen)$;
      if (Exp)
        { $\mathcal{F}_{cc}(\mathbf{C}_1, X, seen \cup FV(\mathbf{Exp}))$; }
      else
        { $\mathcal{F}_{cc}(\mathbf{C}_2, X, seen \cup FV(\mathbf{Exp}))$; }

$\mathcal{F}_{cc}(\mathbf{Exp}, X, seen) =$
  /* emit a line of code for every r.v. U in the corresponding set */
      $\forall\, U \in ((FV(\mathbf{Exp}) \cap \widehat{S}_X) \backslash seen)$ : Cont(U) += X;
      $\forall\, U \in (FV(\mathbf{Exp}) \backslash seen)$ : Ch(U) += X;
```

Figure 4: Transformation rules for outputting inference code with ACU. $FV(\mathbf{Expr})$ denote the free variables in $\mathbf{Expr}$. We assume the switching variables $\widehat{S}_X$ have already been computed. The rules for number statements and case statements are not shown for conciseness—they are analogous to the rules for random variables and if statements respectively.

since $Cont_w(z(d)) = \{x(d)\}$ for any w. The generated code for this process is shown below.

```
void x(d)::add_to_Ch() {
  Cont(z(d)) += x(d);
  Ch(z(d)) += x(d);
  Ch(mu(z(d))) += x(d); }
void z(d)::accept_value(Cluster v) {
// accept the proposed value v for r.v. z(d)
// code for updating references omitted
  for (u in Cont(z(d))) u.del_from_Ch();
  val_z(d) = v;
  for (u in Cont(z(d))) u.add_to_Ch(); }
```

We omit the code for `del_from_Ch()` for conciseness, which is essentially the same as `add_to_Ch()`.

The formal transformation rules for ACU are demonstrated in Fig. 4. $\mathcal{F}_c$ takes in a random variable declaration statement in the BLOG program and outputs the inference code containing methods `accept_value()` and `add_to_Ch()`. $\mathcal{F}_{cc}$ takes in a BLOG expression and generates the inference code inside method `add_to_Ch()`. It keeps track of already seen variables in *seen*, which makes sure that a variable will be added to the contingent set or the children set at most once. Code for removing variables can be generated similarly.

Since we maintain the exact children for each r.v. with ACU, the computation of acceptance ratio α in Eq. 1 for PMH can be simplified to

$$\min\left(1, \frac{|w| \Pr[X = v|w'_{\text{-}X}] \prod_{U \in Ch_{w'}(X)} \Pr[U(w')|w'_{\text{-}U}]}{|w'| \Pr[X = v'|w_{\text{-}X}] \prod_{V \in Ch_w(X)} \Pr[V(w)|w_{\text{-}V}]}\right) \tag{2}$$

Here $|w|$ denotes the total number of random variables existing in w, which can be maintained via RC.

Finally, the computation time of ACU is strictly shorter than that of acceptance ratio α (Eq. 2).

4.2 Implementation of Swift

Lightweight memoization

Here we introduce the implementation details of the memoization code in the *getter* function mentioned in section 4.1.

Objects will be converted to integers in the target code. Swift analyzes the input PP and allocates static memory for memoization in the target code. For open-universe models where the number of random variables are unknown, the *dynamic table* data structure (e.g., `vector` in C++) is used to reduce the amount of dynamic memory allocations: we only increase the length of the array when the number becomes larger than the capacity.

One potential weakness of directly applying a dynamic table is that, for models with multiple nested number statements (e.g. the aircraft tracking model with multiple aircrafts and blips for each one), the memory consumption can be large. In Swift, the user can force the target code to clear all the unused memory every fixed number of iterations via a compilation option, which is turned off by default.

The following code demonstrates the target code for the number variable $\#Ball$ and its associated $color(b)$'s in the urn-ball model (Fig. 2) where `iter` is the current iteration number.

```
vector<int> val_color, mark_color;
int get_color(int b) {
  if (mark_color[b] == iter)
    return val_color[b]; // memoization
  mark_color[b] = iter; // mark the flag
  val_color[b] = ...//sampling code omitted
  return val_color[b];}
int val_num_B, mark_num_B;
int get_num_Ball() {
  if(mark_num_B == iter) return val_num_B;
  val_num_B = ...// sampling code omitted
  if(val_num_B > val_color.size(){//allocate
  { val_color.resize(val_num_B);  //memory
    mark_color.resize(val_num_B);  }
  return val_num_B; }
```

Note that when generating a new PW, by simply increasing the counter `iter`, all the memoization flags (e.g., `mark_color` for $color(\cdot)$) will be automatically cleared.

Lastly, in PMH, we need to randomly select an r.v. to sample per iteration. In the target C++ code, this process is accomplished via polymorphism by declaring an abstract class with a `resample()` method and a derived class for every r.v. in the PP. An example code snippet for the ∞-GMM model is shown below.

```
class MH_OBJ {public: //abstract class
  virtual void resample()=0;//resample step
  virtual void add_to_Ch()=0; //for ACU
  virtual void accept_value()=0;//for ACU
  virtual double get_likeli()=0;
}; // some methods omitted
// maintain all the r.v.s in the current PW
std::vector<MH_OBJ*> active_vars;
```

```
// derived classes for r.v.s in the PP
class Var_mu:public MH_OBJ{public://for mu(c)
  double val; // value for the var
  double cached_val; //cache for proposal
  int mark, cache_mark; // flags
  std::set<MH_OBJ*> Ch, Cont; //sets for ACU
  double getval(){...}; //sample & memoize
  double getcache(){...};//generate proposal
  ... }; // some methods omitted
std::vector<Var_mu*> mu; // dynamic table
class Var_z:public MH_OBJ{ ... };//for z(d)
Var_z* z[20]; // fixed-size array
```

Efficient proposal manipulation

Although one PMH iteration only samples a single variable, generating a new PW may still involve (de-)instantiating an arbitrary number of random variables due to RC or sampling a number variable. The general approach commonly adopted by many PPL systems is to construct the proposed possible world in dynamically allocated memory and then copy it to the current world [Milch *et al.*, 2005a; Wingate *et al.*, 2011], which suffers from significant overhead.

In order to generate and accept new PWs in PMH with negligible memory management overhead, we extend the lightweight memoization to manipulate proposals: Swift statically allocates an extra memoized cache for each random variable, which is dedicated to storing the proposed value for that variable. During resampling, all the intermediate results are stored in the cache. When accepting the proposal, the proposed value is directly loaded from the cache; when rejection, no action is needed due to our memoization mechanism.

Here is an example target code fragment for $mu(c)$ in the ∞-GMM model. `proposed_vars` is an array storing all the variables to be updated if the proposal gets accepted.

```
std::vector<MH_OBJ*>proposed_vars;
class Var_mu:public MH_OBJ{public://for mu(c)
  double val; // value
  double cached_val; //cache for proposal
  int mark, cache_mark; // flags
  double getcache(){ // propose new value
    if(mark == 1) return val;
    if(mark_cache == iter) return cached_val;
    mark_cache = iter; // memoization
    proposed_vars.push_back(this);
    cached_val = ... // sample new value
    return cached_val; }
  void accept_value() {
    val = cached_val;//accept proposed value
    if(mark == 0) { // not instantiated yet
        mark = 1;//instantiate variable
        active_vars.push_back(this);
    } }
  void resample() { // resample
    proposed_vars.clear();
    mark = 0; // generate proposal
    new_val = getcache();
    mark = 1;
    alpha = ... //compute acceptance ratio
    if (sample_unif(0,1) <= alpha) {
      for(auto v: proposed_vars)
        v->accept_value(); // accept
      // code for ACU and RC omitted
    } } }; // some methods omitted
```

Supporting new algorithms

We demonstrate that FIDS can be applied to new algorithms by implementing a translator for the Gibbs sampling [Arora *et al.*, 2010] (Gibbs) in Swift.

Gibbs is a variant of MH with the proposal distribution $g(\cdot)$ set to be the posterior distribution. The acceptance ratio α is always 1 while the disadvantage is that it is only possible to explicitly construct the posterior distribution with conjugate priors. In Gibbs, the proposal distribution is still constructed from the Markov blanket, which again requires maintaining the children set. Hence, FIDS can be fully utilized.

However, different conjugate priors yield different forms of posterior distributions. In order to support a variety of proposal distributions, we need to (1) implement a conjugacy checker to check the form of posterior distribution for each r.v. in the PP at compile time (the checker has nothing to do with FIDS); (2) implement a posterior sampler for every conjugate prior in the runtime library of Swift.

5 Experiments

In the experiments, Swift generates target code in C++ with C++ standard `<random>` library for random number generation and the armadillo package [Sanderson, 2010] for matrix computation. The baseline systems include BLOG (version 0.9.1), Figaro (version 3.3.0), Church (webchurch), Infer.NET (version 2.6), BUGS (winBUGS 1.4.3) and Stan (cmdStan 2.6.0).

5.1 Benchmark models

We collect a set of benchmark models[2] which exhibit various capabilities of a PPL (Tab. 1), including the burglary model (Bur), the hurricane model (Hur), the urn-ball model (U-B(x,y) denotes the urn-ball model with at most x balls and y draws), the TrueSkill model [Herbrich *et al.*, 2007] (T-K), 1-dimensional Gaussian mixture model (GMM, 100 points with 4 clusters) and the infinite GMM model (∞-GMM). We also include a real-world dataset: handwritten digits [LeCun *et al.*, 1998] using the PPCA model [Tipping and Bishop, 1999]. All the models can be downloaded from the GitHub repository of BLOG.

5.2 Speedup by FIDS within Swift

We first evaluate the speedup promoted by each of the three optimizations in FIDS individually.

Dynamic Backchaining for LW

We compare the running time of the following versions of code: (1) the code generated by Swift ("Swift"); (2) the modified compiled code with DB manually turned off ("No DB"), which sequentially samples all the variables in the PP; (3) the hand-optimized code without unnecessary memoization or recursive function calls ("Hand-Opt").

[2]Most models are simplified to ensure that all the benchmark systems can produce an output in reasonably short running time. All of these models can be enlarged to handle more data without adding extra lines of BLOG code.

	Bur	Hur	U-B	T-K	GMM	∞-GMM	PPCA
R				✓	✓	✓	✓
CT		✓	✓		✓	✓	
CC		✓					
OU			✓			✓	
CG	✓	✓	✓		✓	✓	✓

Table 1: Features of the benchmark models. R: continuous scalar or vector variables. CT: context-specific dependencies (contingency). CC: cyclic dependencies in the PP (while in any particular PW the dependency structure remains acyclic). OU: open-universe. CG: conjugate priors.

Alg.	**Bur**	**Hur**	**U-B(20,2)**	**U-B(20,8)**
Running Time (s): Swift v.s. No DB				
No DB	0.768	1.288	2.952	5.040
Swift	0.782	1.115	1.755	4.601
Speedup	0.982	**1.155**	**1.682**	**1.10**
Running Time (s): Swift v.s. Hand-Optimized				
Hand-Opt	0.768	1.099	1.723	4.492
Swift	0.782	1.115	1.755	4.601
Overhead	1.8%	1.4%	1.8%	2.4%
Calls to Random Number Generator				
No DB	$3 * 10^7$	$3 * 10^7$	$1.35 * 10^8$	$1.95 * 10^8$
Swift	$3 * 10^7$	$2.0 * 10^7$	$4.82 * 10^7$	$1.42 * 10^8$
Calls Saved	0%	33.3%	64.3%	27.1%

Table 2: Performance on LW with 10^7 samples for Swift, the version without DB and the hand-optimized version

We measure the running time for all the 3 versions and the number of calls to the random number generator for "Swift" and "No DB". The result is concluded in Tab. 2.

The overhead due to memoization compared against the hand-optimized code is less than 2.4%. We can further notice that the speedup is proportional to the number of calls saved.

Reference Counting for PMH

RC only applies to open-universe models in Swift. Hence we focus on the urn-ball model with various model parameters. The urn-ball model represents a common model structure appearing in many applications with open-universe uncertainty. This experiment reveals the potential speedup by RC for real-world problems with similar structures. RC achieves greater speedup when the number of balls and the number of observations become larger.

We compare the code produced by Swift with RC ("Swift") and RC manually turned off ("No RC"). For "No RC", we traverse the whole dependency structure and reconstruct the dynamic slice every iteration. Fig. 5 shows the running time of both versions. RC is indeed effective – leading to up to 2.3x speedup.

Adaptive Contingency Updating for PMH

We compare the code generated by Swift with ACU ("Swift") and the manually modified version without ACU ("Static Dep") and measure the number of calls to the likelihood function in Tab. 3.

The version without ACU ("Static Dep") demonstrates the best efficiency that can be achieved via compile-time analysis without maintaining dependencies at runtime. This version statically computes an upper bound of the children set by $\widehat{Ch}(X) = \{Y : X \text{ appears in } C_Y\}$ and again uses Eq.(2) to compute the acceptance ratio. Thus, for models with fixed dependencies, "Static Dep" should be faster than "Swift".

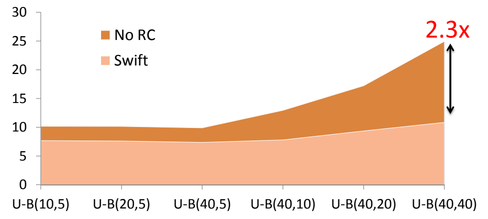

Figure 5: Running time (s) of PMH in Swift with and without RC on urn-ball models for 20 million samples.

Alg.	**Bur**	**Hur**	**U-B(20,10)**	**U-B(40,20)**
Running Time (s): Swift v.s. Static Dep				
Static Dep	0.986	1.642	4.433	7.722
Swift	0.988	1.492	3.891	4.514
Speedup	**0.998**	**1.100**	**1.139**	**1.711**
Calls to Likelihood Functions				
Static Dep	$1.8 * 10^7$	$2.7 * 10^7$	$1.5 * 10^8$	$3.0 * 10^8$
Swift	$1.8 * 10^7$	$1.7 * 10^7$	$4.1 * 10^7$	$5.5 * 10^7$
Calls Saved	**0%**	**37.6%**	**72.8%**	**81.7%**

Table 3: Performance of PMH in Swift and the version utilizing static dependencies on benchmark models.

In Tab. 3, "Swift" is up to 1.7x faster than "Static Dep", thanks to up to 81% reduction of likelihood function evaluations. Note that for the model with fixed dependencies (Bur), the overhead by ACU is almost negligible: 0.2% due to traversing the hashmap to access to child variables.

5.3 Swift against other PPL systems

We compare Swift with other systems using LW, PMH and Gibbs respectively on the benchmark models. The running time is presented in Tab. 4. The speedup is measured between Swift and the fastest one among the rest. An empty entry means that the corresponding PPL fails to perform inference on that model. Though these PPLs have different host languages (C++, Java, Scala, etc.), the performance difference resulting from host languages is within a factor of 2[3].

5.4 Experiment on real-world dataset

We use Swift with Gibbs to handle the probabilistic principal component analysis (PPCA) [Tipping and Bishop, 1999], with real-world data: 5958 images of digit "2" from the hand-written digits dataset (MNIST) [LeCun *et al.*, 1998]. We compute 10 principal components for the digits. The training and testing sets include 5958 and 1032 images respectively, each with 28x28 pixels and pixel value rescaled to $[0, 1]$.

Since most benchmark PPL systems are too slow to handle this amount of data, we compare Swift against other two

[3]`http://benchmarksgame.alioth.debian.org/`

PPL	Bur	Hur	U-B (20,10)	T-K	GMM	∞-GMM
LW with 10^6 samples						
Church	9.6	22.9	179	57.4	1627	1038
Figaro	15.8	24.7	176	48.6	997	235
BLOG	8.4	11.9	189	49.5	998	261
Swift	0.04	0.08	0.54	0.24	6.8	1.1
Speedup	**196**	**145**	**326**	**202**	**147**	**214**
PMH with 10^6 samples						
Church	12.7	25.2	246	173	3703	1057
Figaro	10.6	-	59.6	17.4	151	62.2
BLOG	6.7	18.5	30.4	38.7	72.5	68.3
Swift	0.11	0.15	0.4	0.32	0.76	0.60
Speedup	**61**	**121**	**75**	**54**	**95**	**103**
Gibbs with 10^6 samples						
BUGS	87.7	-	-	-	84.4	-
Infer.NET	1.5	-	-	-	77.8	-
Swift	0.12	0.19	0.34	-	0.42	0.86
Speedup	**12**	∞	∞	-	**185**	∞

Table 4: Running time (s) of Swift and other PPLs on the benchmark models using LW, PMH and Gibbs.

scalable PPLs, Infer.NET and Stan, on this model. Both PPLs have compiled inference and are widely used for real-world applications. Stan uses HMC as its inference algorithm. For Infer.NET, we select variational message passing algorithm (VMP), which is Infer.NET's primary focus. Note that HMC and VMP are usually favored for fast convergence.

Stan requires a tuning process before it can produce samples. We ran Stan with 0, 5 and 9 tuning steps respectively[4]. We measure the perplexity of the generated samples over test images w.r.t. the running time in Fig. 6, where we also visualize the produced principal components. For Infer.NET, we consider the mean of the approximation distribution.

Swift quickly generates visually meaningful outputs with around 10^5 Gibbs iterations in 5 seconds. Infer.NET takes 13.4 seconds to finish the first iteration[5] and converges to a result with the same perplexity after 25 iterations and 150 seconds. The overall convergence of Gibbs w.r.t. the running time significantly benefits from the speedup by Swift.

For Stan, its no-U-turn sampler [Homan and Gelman, 2014] suffers from significant parameter tuning issues. We also tried to manually tune the parameters, which does not help much. Nevertheless, Stan does work for the simplified PPCA model with 1-dimensional data. Although further investigation is still needed, we conjecture that Stan is very sensitive to its parameters in high-dimensional cases. Lastly, this experiment also suggests that those parameter-robust inference engines, such as Swift with Gibbs and Infer.NET with VMP, would be preferred at practice when possible.

[4]We also ran Stan with 50 and 100 tuning steps, which took more than 3 days to finish 130 iterations (including tuning). However, the results are almost the same as that with 9 tuning steps.

[5]Gibbs by Swift samples a single variable while Infer.NET processes all the variables per iteration.

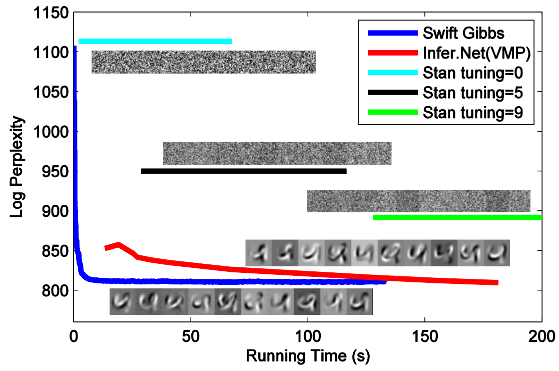

Figure 6: Log-perplexity w.r.t running time(s) on the PPCA model with visualized principal components. Swift converges faster.

6 Conclusion and Future Work

We have developed Swift, a PPL compiler that generates model-specific target code for performing inference. Swift uses a dynamic slicing framework for open-universe models to incrementally maintain the dependency structure at runtime. In addition, we carefully design data structures in the target code to avoid dynamic memory allocations when possible. Our experiments show that Swift achieves orders of magnitudes speedup against other PPL engines.

The next step for Swift is to support more algorithms, such as SMC [Wood *et al.*, 2014], as well as more samplers, such as the block sampler for (nearly) deterministic dependencies [Li *et al.*, 2013]. Although FIDS is a general framework that can be naturally extended to these cases, there are still implementation details to be studied.

Another direction is partial evaluation, which allows the compiler to reason about the input program to simplify it. Shah *et al.* [2016] proposes a partial evaluation framework for the inference code (P_I in Fig. 1). It is very interesting to extend this work to the whole Swift pipeline.

Acknowledgement

We would like to thank our anonymous reviewers, as well as Rohin Shah for valuable discussions. This work is supported by the DARPA PPAML program, contract FA8750-14-C-0011.

References

[Agrawal and Horgan, 1990] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[Arora *et al.*, 2010] Nimar S. Arora, Rodrigo de Salvo Braz, Erik B. Sudderth, and Stuart J. Russell. Gibbs sampling in open-universe stochastic languages. In *UAI*, pages 30–39. AUAI Press, 2010.

[Chaganty *et al.*, 2013] Arun Chaganty, Aditya Nori, and Sriram Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS*, pages 153–160, 2013.

[Cytron *et al.*, 1989] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[Goodman *et al.*, 2008] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *UAI*, pages 220–229, 2008.

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

[Homan and Gelman, 2014] Matthew D Homan and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *The Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

[Hur *et al.*, 2014] Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. Slicing probabilistic programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 133–144, New York, NY, USA, 2014. ACM.

[Kazemi and Poole, 2016] Seyed Mehran Kazemi and David Poole. Knowledge compilation for lifted probabilistic inference: Compiling to a low-level language. 2016.

[Kucukelbir *et al.*, 2015] Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. Automatic variational inference in Stan. In *NIPS*, pages 568–576, 2015.

[LeCun *et al.*, 1998] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[Li *et al.*, 2013] Lei Li, Bharath Ramsundar, and Stuart Russell. Dynamic scaled sampling for deterministic constraints. In *AISTATS*, pages 397–405, 2013.

[Lunn *et al.*, 2000] David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS - a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, October 2000.

[Mansinghka *et al.*, 2013] Vikash K. Mansinghka, Tejas D. Kulkarni, Yura N. Perov, and Joshua B. Tenenbaum. Approximate Bayesian image interpretation using generative probabilistic graphics programs. In *NIPS*, pages 1520–1528, 2013.

[McAllester *et al.*, 2008] David McAllester, Brian Milch, and Noah D Goodman. Random-world semantics and syntactic independence for expressive languages. Technical report, 2008.

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

[Milch *et al.*, 2005a] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[Milch *et al.*, 2005b] Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L. Ong, and Andrey Kolobov. Approximate inference for infinite contingent Bayesian networks. In *Tenth International Workshop on Artificial Intelligence and Statistics, Barbados*, 2005.

[Minka *et al.*, 2014] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014.

[Nori *et al.*, 2014] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence*, 2014.

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

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

[Plummer, 2003] Martyn Plummer. 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.

[Ritchie *et al.*, 2016] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *AISTATS 2016*, 2016.

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

[Shah *et al.*, 2016] Rohin Shah, Emina Torlak, and Rastislav Bodik. SIMPL: A DSL for automatic specialization of inference algorithms. *arXiv preprint arXiv:1604.04729*, 2016.

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

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

[Tristan *et al.*, 2014] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. Augur: Data-parallel probabilistic modeling. In *NIPS*, pages 2600–2608. 2014.

[Wingate *et al.*, 2011] David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

[Wood *et al.*, 2014] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 2–46, 2014.

[Wu *et al.*, 2014] Yi Wu, Lei Li, and Stuart J. Russell. BFiT: From possible-world semantics to random-evaluation semantics in open universe. In *Neural Information Processing Systems, Probabilistic Programming workshop*, 2014.

[Yang *et al.*, 2014] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating efficient MCMC kernels from probabilistic programs. In *AISTATS*, pages 1068–1076, 2014.