Incrementalizing MCMC in Probabilistic Programs Through Tracing and Slicing

Lingfeng Yang
Stanford University
lyang@cs.stanford.edu

Yi-Ting Yeh
Stanford University
yitingy@stanford.edu

Noah D. Goodman
Stanford University
ngoodman@stanford.edu

Pat Hanrahan
Stanford University
hanrahan@cs.stanford.edu

Abstract

Probabilistic programming enables high-level specification of complex probabilistic models without the need for hand-built inference techniques. We present a dynamic compilation technique for increasing the efficiency of Markov Chain Monte Carlo (MCMC) inference on probabilistic programs. We focus on Church, a probabilistic extension of untyped call-by-value lambda calculus. MCMC in this setting is a random walk over program paths; it is extremely expensive because each iteration of MCMC requires a complete execution of the program with side computations that track random choices and probabilities. However, there exist many opportunities to remove redundant computation. In particular, we exploit the fact that only some random choices influence control flow. Partially evaluating these structural choices away results in a trace that can be optimized via existing techniques from JIT compilers and incremental computation. This minimizes the amount of work done at each step in the random walk. We evaluate this technique on several representative probabilistic models, finding that we achieve orders of magnitude speedup compared with unoptimized Church MCMC. Furthermore, our technique becomes competitive with inference engines for more specialized and less expressive statistical languages.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Machine learning has made itself a part of our everyday lives. Internet search results rely on ranking algorithms. Photos are automatically adjusted based on face recognition routines. Our mail gets to the right destination, despite the handwriting skill of the sender, thanks to optical character recognition.

PLDI '2013 date, City.

Machine learning fundamentally treats the computer as a rational agent, capable of reasoning under uncertainty. Probability theory thus provides the foundation for many machine learning systems in wide use. We use it to formulate "fuzzy" problems such as "which character did the user mean to write" or "what movies will this user like given his viewing history" in a precise way, enabling us to construct and compose such problems together. In this way, probability distributions, Bayes' rule, and other elements of probability theory are tools that we use to simulate the world while taking into account uncertainty.

However, actually formulating probabilistic models and performing inference is not a simple task. Suppose we are modeling the trajectories of aircraft in a certain airspace. We have only noisy radar blips as observations, and we would like to infer the location of each aircraft. Before even considering the implementation details, we need to design the model: What are the random variables? Do we assume that there is a fixed number of aircraft at any time, or can it vary with the number of blips? Do we explicitly model the presence of weather effects? It is often difficult to answer these questions without having tried the possible alternatives. Yet, implementing even just one model in this setting and performing inference can be quite involved. Most of the time, just the inference algorithm is implemented, and it is specialized to a particular model. This practical situation is very far from the ideal where we can simply tell the computer, $P(\text{aircraft presence}|\text{radar blips})$ and get an answer.

To address this, recently there has been much work done in *probabilistic programming languages*, which provide an intuitive abstraction with which models can be quickly formulated and prototyped, along with built-in inference algorithms that do not require the user to implement—or even understand—sophisticated probabilistic inference algorithms. The essential observation is that probability distributions can be represented by programs with random choice primitives; inference can be specified by a query function that expresses posterior probabilities. Indeed the execution of a general-purpose program with random choices induces a distribution over return values in a very precise sense [Ramsey and Pfeffer 2002, Goodman et al. 2008], and furthermore any computable distribution can be so represented [Freer and Roy 2010].

In this paper, we focus on Church [Goodman et al. 2008], an extension of Scheme with random choice and query primitives. Inference in Church is usually accomplished through Markov Chain Monte Carlo (MCMC), and in particular the Metropolis-Hastings

(MH) algorithm. MH is based on perturbing the set of random choices made during program execution (a *proposal*), then accepting or rejecting the perturbation according to an *acceptance probability*. Iterating this process results in a sequence of samples, each of which is a stochastic perturbation of the previous one. The collection of samples in the sequence will asymptotically reflect the true distribution—but millions of samples can be required for non-trivial models. Therefore, in order to perform MH efficiently, it is important to make each stochastic perturbation (proposal) and acceptance probability computation as fast as possible. When the model is specified as a program, the problem is one of *incrementalization*: minimizing redundancy in successive proposal and acceptance computations. We describe techniques for incrementalization that show the following performance improvements:

1. MH over probabilistic programs can be accelerated by compiling away *structural choices*: the random choices that affect control flow. This is done using a just-in-time tracing technique, producing straight-line traces of the acceptance ratio calculation specialized to a particular set of structural choices. Optimizations such as dead code elimination and allocation removal are applied to the trace, resulting in order-of-magnitude speedups.
2. Incrementalization, or avoiding the rerunning of parts of the trace that do not depend on the recently perturbed variable, can lead to speedups in models where there are many independent computations on random choices.
3. The sum total of the optimizations brings an implementation of the Church probabilistic programming language in the same realm of performance as a more restricted statistical package, JAGS.

This paper begins by giving background on Church and MCMC. We then give an overview of the techniques we use and details of how we adapt these tracing and slicing techniques to our setting. Finally, we give performance comparisons with unoptimized Church, JAGS, and hand-coded inference, and conclude.

2. Background

Probabilistic programming. We will focus on the probabilistic programming language Church, a probabilistic extension of call-by-value lambda calculus. A Church program includes calls to random choice primitives, which are functions that output randomly sampled values. Each call corresponds to a random variable in the mathematical sense [Goodman et al. 2008]. Because Church programs may be recursive, the set of random variables need not be fixed. In general, the set of future random choices made depend on previous random choices.

Inference in a Church program is specified by the `query` function. This function can be specified in Church (as the rejection sampling process), but the main problem for implementations is to provide a more efficient implementation of `query`—here we explore an efficient MH version of `query`. For convenience we augment Church with `factor` statements—each call to factor multiplies the probability of the current sample by its argument. That is, `factor` provides a convenient way to specify a soft constraint. One can think of `factor` as a soft version of `assert` statements found in constraint programming languages like SMT-LIB 2.0 [Barrett et al. 2010].

For intuition, consider the following Church program, which results in lists of binary numbers of varying length; the lists are weighted through factor calls to favor lists in which consecutive pairs of numbers are equal.

```
1  (query num-samples
2    (let*
3      ([n (randint 2 4)]
4       [xs (repeat n (lambda () (randint 0 1)))]
5       [eq (lambda (x y)
6             (factor (if (= x y) 0.0 (log 0.1))))]
7       [constr (map
8                 (lambda (xy) (eq (car xy) (car (cdr xy))))
9                 (bigram xs))])
10     xs))
```

The execution of the `let*` expression proceeds as follows. We first sample the number of elements in the list (line 3) through the use of the `randint` sampling function, which takes an upper and lower bound as arguments and returns an integer uniformly distributed in the bounds (inclusive). Thus, in this program, the list can be 2, 3, or 4 elements long. The list is constructed with the call to `repeat` on line 4 which takes the number we sampled in line 3 as the desired length of the list along with a function that is run to produce each element. `xs` is now a list of random length whose entries are random binary values. Then, at line 5, we define a soft constraint using `factor`. `eq` softly constrains pairs of numbers to be equal. It multiplies the probability of the current sample by 0.1 if its arguments are unequal, and otherwise leaves the probability the same. Note that computations are done using log probabilities to avoid underflow. Finally, at line 6, we apply this factor function to all consecutive pairs of elements in the list through the use of `map` and `bigram`. `xs` is returned at line 7.

To what probability distribution does this correspond? Let n represent the sampled value of `n`, f the `eq` factor function and $x_1, \ldots x_n$ `xs`. The probability is then

$$P(x_1, \ldots, x_n) \propto p(n) \prod_{i=1}^{n} p(x_i) \prod_{i=1}^{n-1} f(x_i, x_{i+1}). \qquad (1)$$

where $p(n)$ denotes the probability of drawing the sampled value of n and $p(x_i)$ denotes the probability of drawing that particular binary number x_i in the list. Every random choice made in the program corresponds to a mathematical random variable. Hence, the RHS includes a term for each random choice made throughout the program and for each call to a soft constraint defined by `factor`. Critically, the range of the product can depend on control flow of the program, and the arguments to the probability functions depend on the computations done by the program— thus this simple-looking formula in fact abstracts over program execution.

Markov Chain Monte Carlo. Abbreviated "MCMC", Markov Chain Monte Carlo is a technique for estimating probabilities. We will focus on one of the most basic variants, Metropolis-Hastings (MH).

The Metropolis-Hastings algorithm [Hastings 1970] takes a possibly unnormalized probability density $p(\cdot)$ as input and returns a series of states x_n that are distributed approximately according to $p(\cdot)$. It does so by repeatedly transforming the *state* x using a stochastic update procedure. This procedure favors some transitions between states over others, in such a way that the states are (asymptotically) visited in proportion to the input distribution.

The transformation itself consists of two steps:

1. **Proposal:** Perturbing x according to a simple *proposal* distribution $q(x'|x)$, to obtain a transformed state x'.
2. **Acceptance:** Deciding whether the next state is x' (*accept*) or remains x (*reject*) by taking a sample α from the unit interval and comparing it to

$$\min\left\{1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right\}.$$

If α is less than that amount we accept, otherwise we reject. (The $q(x'|x)$ and $q(x|x')$ factors serve to compensate for bias in the random walk.)

MH over probabilistic programs. We follow the lightweight MH implementation scheme given in [Wingate et al. 2011-1]. Consider the Church program given above. First, we run the expression inside `query` and record the resulting random choices in a side computation. This results in a sample along with a database that tracks which choice was made for each call to a random choice function. The MH proposal is made by perturbing a single choice in the database then re-running the program, making sure to re-use choices stored in the database when possible, but resampling and storing any new choices that may occur. The $p(x)$ and $p(x')$ parts of the acceptance ratio are computed according to $P(x_1, \ldots x_n)$ given above, and the $q(x|x'), q(x'|x)$ parts are computed based on the initial perturbation and the total probability of the choices that have been added or removed.

In order to identify choices that may be re-used, and which choices have been added or removed, we require a way to name each random choice consistently. The naming (or *addressing*) scheme from [Wingate et al. 2011-1] is based on the stack trace of function calls at the point the choice is invoked. However, addressing can be expensive, as it requires the computation of what is essentially a stack trace at each random choice. One of the computations we can often compile away using tracing JIT techniques is this addressing. In particular, if the set of random variables is fixed at the next iteration, their addresses do not need to be recomputed.

Tracing JIT. We propose to accelerate the MH process on probabilistic programs by using a tracing JIT compiler. Tracing JIT compilers have a long history [Aycock 2003]. They generally operate by instrumenting an existing interpreter for a dynamic language, recording all the calls to primitive functions within some defined boundary. For example, some tracing JIT compilers will store traces for each method call in an object-oriented language, while others will store traces for loops. Because the code inside such methods and loops will be executed very often, it is worthwhile to generate a trace and optimize the result. Once a trace has been recorded many possible optimizations one can be applied. For instance, constant folding is obtained by tracing itself. Other optimizations include the removal of run-time type checks and unboxing of primitive types [Gal et al 2009] along with allocation removal [Bolz et al. 2011]. In our setting, we store traces for the body of a probabilistic program running MH—a loop where the exact same thing is computed each iteration, modulo random choices. We may then bring the full weight of optimizations of tracing JIT compilers to bear on the MH loop. We will explain our tracing setup in detail in Section 4.

Incremental computation. Incremental computation is a subdiscipline that concerns itself with the following problem: Given a function f, previous input x, previous output $y = f(x)$, and new input x', how to calculate $y' = f(x')$ as efficiently as possible? It is also known as *self-adjusting computation* [Acar 2005]. In our setting, we would like to answer this question when x and x' are successive states of MH and f is the acceptance ratio as expressed through the semantics of the probabilistic program.

Consider the probability score associated with a run of the program, given above. This will be included in both the numerator and denominator of the acceptance ratio. Some factors will cancel out, as the same value will be computed for them before and after the perturbation. In performance-critical settings, MCMC is incrementalized by hand through exploiting this fact. We show that in our setting, we may obtain this incrementalization automatically through relatively simple techniques such as reaching-definitions analysis on the traces we record. We give a detailed explanation of our incrementalization scheme in Section 5.

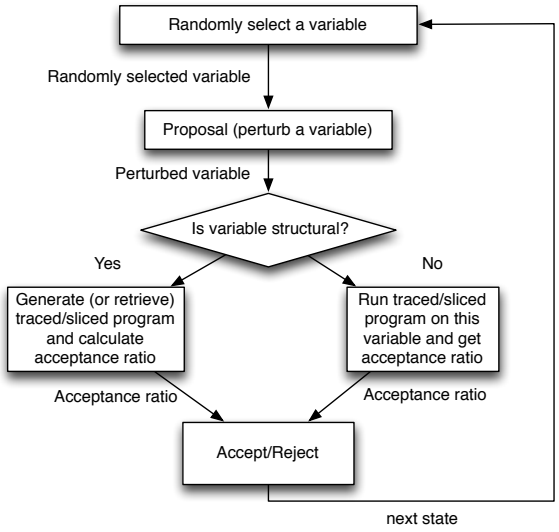

Figure 1. A high level overview of the MCMC loop as implemented by our system.

3. Overview

Many MH proposals will cause no change in the control flow of the program. In order to exploit this structure, we distinguish two types of random variables, *structural* and *non-structural*. Structural variables affect the control flow of the program, while non-structural variables cannot. In the example in the last section, the number of elements in the list, `n`, is a structural variable, and each binary number inside the list, `xs`, is a non-structural variable. For simplicity we provide a structural and non-structural version of each random sampling primitive and require the user to choose the appropriate version; a control-flow analysis could in principle be used to infer which variables are (non-)structural.

Figure 1 describes the high-level structure of our system. During each iteration, we randomly select a variable and perturb its value. If it is a structural variable, we generate or retrieve a trace (explained in Section 4). If it is not, we re-run the current trace incrementally with the new value (explained in Section 5), which is faster. Then, we accept or reject the current sample and repeat the process.

Figure 2 illustrates three different scenarios. In the first iteration, we have performed a change to a structural variable, changing `n` from 2 to 3. If this is the first time that `n` equals 3, we create and run a new trace $T(R_1, R_2, R_3)$. Now, assume that the proposal is accepted. In the next iteration, we propose a change to R_2, which is a non-structural variables. Our system then runs $T(R_1, R_2, R_3)$ *incrementally* as if it were only a function of R_2, remembering the parts of the state having to do with R_1 and R_3. Finally, in the third iteration, we propose another change to the structural variable `n`, changing it from 3 to 2. In this case, we retrieve the trace for `n = 2` from the cache, and compute the acceptance ratio.

4. Tracing

The goal of tracing is to optimize the acceptance ratio calculation, specializing it to a particular set of structural choices. The traces produced are functions that take an assignment to the remaining non-structural choices as input and output the unnormalized prob-

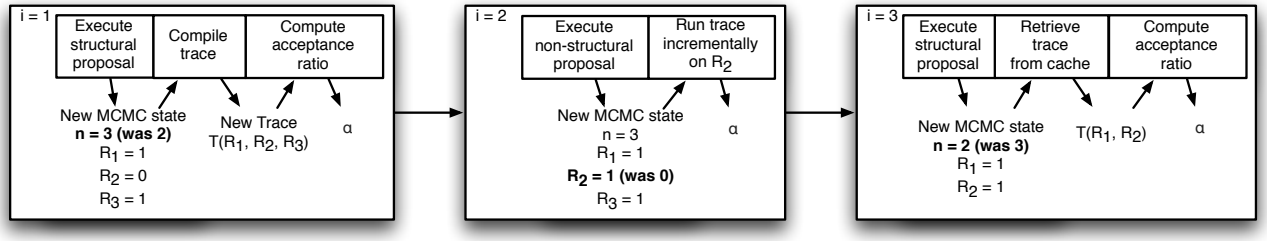

Figure 2. The situation of having 3 accepted MCMC iterations where the first and last proposals are structural and the second is nonstructural. We compile or retrieve a trace for every structural proposal, and when there is a nonstructural proposal, we many incrementally execute the current trace to more quickly calculate the acceptance ratio.

ability score corresponding to these choices, forming the numerator or denominator of the acceptance ratio. They also compute the sampled value of the program. We then further optimize these straight-line traces by applying standard techniques such as dead code elimination and allocation removal.

Generation of traces. We need to compute both the sample and the corresponding unnormalized score. Thus, our primitive operations include not only the usual arithmetic operators and list constructors/destructors prevalent in R5RS Scheme, we also include the `factor` and `xrp+score` primitives. `factor` is used to explicitly contribute terms to the acceptance ratio as a result of evaluating soft constraints. We trace a call to `factor` for each call to `factor` in the original program. `xrp+score` is used to track the contribution of each random choice to the acceptance ratio. It takes a scoring function, sample, and parameters as arguments, returning the sample directly. However, it has the side effect of running the scoring function on the sample and parameters, then contributing the result to the acceptance ratio. We trace a call to `xrp+score` whenever we make a random choice in the program.

For example, tracing the running example from Section 2 for `n = 3` results in this series of statements, after performing dead code elimination and allocation removal (two known optimizations for linear traces):

```
(define X0 (xrp+score randint-scorer R0 0 1))
(define X1 (xrp+score randint-scorer R1 0 1))
(define X2 (xrp+score randint-scorer R2 0 1))
(define X3 (factor (eq 1.0 X0 X1)))
(define X4 (factor (eq 1.0 X1 X2)))
(define L0 (cons X2 '()))
(define L1 (cons X1 L0))
(define L2 (cons X0 L1))
```

Note that we distinguish functions that are used purely as factors and do not trace into them to avoid unnecessary tracing—for example, we did not trace out the `eq` function, even though it is not a primitive. Assigning `R0, R1, R2` to particular values and running this trace calculates the total unnormalized probability and leaves the sampled value in `L2`. The acceptance ratio can then be calculated by comparing the resulting score against another obtained by running the trace on a different assignment to `R0, R1, R2`.

5. Incrementalization

Each trace thus represents a complete computation of the numerator or denominator of the acceptance ratio. We would like to perform the minimal computation possible between perturbations. This is the goal of incrementalization. For instance, if we perturbed `R2` in the above trace, we would only have to re-compute `X2` and `X4`. The corresponding acceptance ratio will have as numerator the score after perturbing `R2`, and the score before perturbation as denominator. The factors associated with `X0, X1,` and `X3` will cancel out from this ratio; the computation can be performed very cheaply.

Slicing by reaching definitions. We will refer to the subset of variables that need recomputation as a *slice*. We compute each slice by performing a reaching definitions analysis [Aho et al. 2006] on the trace. Given an input variable `Ri`, we identify all definitions that variable reaches by constructing a least fixed-point. We introduce one modification: input variables may reach the definition of other input variables, but no further. This is because of an invariant of the `xrp+score` function; it always returns exactly the value of its second argument, which represents the sampled value of a random choice. For example, suppose we were calculating the slice for `R0` in this trace:

```
(define Y0 (xrp+score randint-scorer R0 0 1))
(define Y1 (xrp+score randint-scorer R1 Y0 2))
(define Y2 (+ Y1 5))
...
```

The input variable `R0` reaches the definition of `Y1`, but in the final slice computation, we assume it does not reach any further, namely, it does not reach `Y2`. The final slices would then be:

```
;; R0 slice
(set! Y0 (xrp+score randint-scorer R0 0 1))
(set! Y1 (xrp+score randint-scorer R1 Y0 2))

;; R1 slice
(set! Y1 (xrp+score randint-scorer R1 Y0 2))
(set! Y2 (+ Y1 5))
```

Caching with closures. One cannot simply run these slices in isolation; they must refer to a consistent state. For instance, in the slice for `R1`, when we run `(set! Y1 (xrp+score randint-scorer R1 Y0 2))`, we must use the correct value of `Y0`.

Our solution is to generate closures for each slice with an enclosing scope where the full set of variables is defined, i.e.,

```
;; Full trace
(lambda (R0 R1 R2)
  (define score 0.0)
  (define X0 (xrp+score randint-scorer R0 0 1))
  ...
  (define X3 (factor (eq 1.0 X0 X1)))
  ...
  (define L2 (cons X0 L1))

  (list
    ;; Slices

    (lambda (R0)
      (begin
        (set! score 0.0)
        ...
        (set! X0 (xrp+score randint-scorer R0 0 1))
        ...
        score))
    ...))
```

Now consider what happens when we run one slice after another. As we run the statements in each slice, the variables all reference the most recently computed value. Incrementalization often involves elaborate caching schemes; we achieve caching implicitly by employing closures.

Example. To give the big picture, below is an overview of what happens to a trace (optimization and incrementalization) generated by the running example from Section 2:

```
;; Trace in full.

V0 = (xrp+score randint-scorer R0 0 1)
V1 = (xrp+score randint-scorer R1 0 1)
V2 = (xrp+score randint-scorer R2 0 1)
V3 = (cons V2 '())
V4 = (cons V1 V3)
V5 = (cons V0 V4)
V6 = (cdr V5)
V7 = (null? V6)
:
:
V34 = (car V33)
V35 = (factor (eq 1.0 V32 V34))
V36 = (cdr V28)
:
:
V42 = (car V41)
V43 = (factor (eq 1.0 V40 V42))
V44 = (cdr V36)
:

;; Trace after optimizations
;; (dead code elimination, allocation removal).

X0 = (xrp+score randint-scorer R0 0 1)
X1 = (xrp+score randint-scorer R1 0 1)
X2 = (xrp+score randint-scorer R2 0 1)
X3 = (factor (eq 1.0 X0 X1))
X4 = (factor (eq 1.0 X1 X2))
L0 = (cons X2 '())
L1 = (cons X1 L0)
L2 = (cons X0 L1)

;; Slices for incremental score computation.

;; R0

X0 = (xrp+score randint-scorer R0 0 1)
X3 = (factor (eq 1.0 X0 X1))
(set-car! L2 X0)

;; R1

X1 = (xrp+score randint-scorer R1 0 1)
X3 = (factor (eq 1.0 X0 X1))
X4 = (factor (eq 1.0 X1 X2))
(set-car! L1 X1)

;; R2

X2 = (xrp+score randint-scorer R2 0 1)
X4 = (factor (eq 1.0 X1 X2))
(set-car! L0 X2)
```

6. Caching and state restoration

When encountering a new structural state we must compile not just the individual statements in the trace of the probabilistic program, but also resampling, proposal, and scoring functions. When incrementalization is activated we also compile slices. This can be a very expensive process, especially when considering the various analyses and optimizations that are applied. We thus require a way to minimize the amount of repetitive tracing: at a high level we simply cache compiled traces and associated code.

Let's go back to the running example in Section 2. Consider the number of elements in the list, `n`. Here, n can be one of $\{2, 3, 4\}$. Each n will correspond to a different trace $p_n(x_1, \ldots x_n)$— traces that, due to the self-contained and pure-functional nature of the program modulo random choices, need not be re-compiled every time we revisit a particular n. We thus store each trace in a cache where the keys are the values of the structural random variables involved. This is crucial for obtaining reasonable performance in models with structure change; the results section will contain more on this.

However, a complication arises if caching is used. Suppose we are in the process of making a structural change. Let x_{S_1} and x_{S_2} refer to two assignments to structural variables, and assume we are proposing x_{S_2}. Let x_{N_1} and x_{N_2} be the respective, corresponding non-structural components. There may be a different number of components in x_{N_1} versus x_{N_2}.

If we do not cache, we re-trace the probabilistic program, sampling the components in x_{N_2} that are new, and copying components from x_{N_1} if we have seen them before as described in [Wingate et al. 2011-1]. We also need to track which variables have been added/removed, in order to compute the proposal forward/backward correction

$$\frac{\prod_{i=1}^{|B|} p(x_{b_i}|x_1 \ldots x_{b_i-1})}{\prod_{i=1}^{|F|} p(x_{f_i}|x_1 \ldots x_{f_i-1})},$$

where the set $B = \{x_{b_i}\}$ refers to the variables in x_{N_1} but not in x_{N_2}, and the set $F = \{x_{f_i}\}$ refers to variables in x_{N_2} but not in x_{N_1}.

However, if we use caching and x_{S_2} has been seen before, we only get back a trace $p_{S_2}(x_{N_2})$. How do we determine which components of x_{N_1} to keep in x_{N_2}, and how do we compute x_{b_i} and x_{f_i}?

Structure change function. We address this through additionally compiling (and caching) a *structure change function* $C_{S_1,S_2}(\cdot)$ for each structural proposal. There is one such function for each pair of assignments to structural variables. It takes as input a vector of non-structural values x_{N_1} and outputs the new vector of non-strucutral values x_{N_2}, so that values in x_{N_1} are mapped to their corresponding positions in x_{N_2}. If there are non-structural values in x_{N_2} that do not occur in x_{N_1}, it will resample those values from the prior. This is conveyed through a special placeholder value, `'new`. We use the addresses of the variables to determine how this mapping occurs; the i-th component of x_{N_1} maps to the j-th component of x_{N_2} if and only if they have the same address.

For example, the structure change function corresponding to the simple `repeat` example for $n = 3$ to $n = 4$ would be `(lambda (X0 X1 X2)(list X0 X1 X2 'new))`. Conversely, the structure change function for $n = 4$ to $n = 3$ is `(lambda (X0 X1 X2 X3)(list X0 X1 X2))`.

By running the structure change function on x_{N_1}, we can determine which components of x_{N_2} taken from x_{N_1} and which components are resampled. x_{N_2} is then a valid input to its corresponding trace. The structure change function also gives us the set of newly created choices x_{f_i}, and by compiling and running the reverse structure change function we can determine x_{b_i}. We then use this information to calculate the MH correction, using the individual scoring functions for choices in x_{f_i} and x_{b_i} (explained below).

Resampling, proposal, and scoring thunks. During tracing, whenever we encounter a non-structural random variable, we produce a variable that is one of the parameters to $p_n(x_1 \ldots x_n)$. However, this is not enough. It is also necessary to be able to quickly re-sample each x_i conditioned on the values of $x_1 \ldots x_{i-1}$ and to compute the factor in the probability solely due to that x_i (such as for computing reversible jump corrections). To this end, we also compile *resampling*, *scoring*, and *proposal* functions r_i, s_i, q_i for each x_i. Each one is a thunk that returns the appropriate value given the other variables. For example, suppose we have the following

trace, where the first sampled variable `X0` is used as a parameter for the second:

```
(define X0 (xrp+score uniform-scorer R0 0 1))
(define X1 (xrp+score gaussian R1 X0))
```

We would compile resampling functions for `X0` and `X1`: `(lambda ()(uniform 0 1))`, and `(lambda ()(gaussian X0))` respectively. Note the use of `X0` in the second function; like the slices, they are put in an enclosing scope where these variables are visible. The same technique—compiling thunks—may be used to generate functions that run proposals $q(x_i'|x_i, x_1 \dots x_{i-1})$ and compute the forward probability $p(x_i|x_1 \dots x_{i-1})$ for each variable. We compile all three functions —resampling, proposals, and forward probabilities— as part of each trace.

7. Results

To sum up, our technique is based on tracing the repeated evaluation of the probabilistic program inherent in the MH algorithm. Each trace represents a different path in control flow through the program—a path that can be associated to particular values of the structural variables. We optimize this trace, removing unused statements, allocations, and lookups. The final trace is then incrementalized by slicing out, for each nonstructural variable, the subset of statements that need re-computation upon each variable's perturbation. Through the trace caching mechanism, already-compiled traces may be reused whenever the model state involves the same set of non-structural variables.

We now evaluate our method with a series of benchmark problems demonstrating its advantages and tradeoffs. We must evaluate two main issues:

- Is tracing useful? Tracing will remove certain repetitive operations, but by how much can this actually increase performance, not considering overhead? As we increase the complexity of the model (by increasing the number of variables), how does tracing scale? Does incrementalization by slicing exploit independence structure of the model, as intended?
- How high is the overhead? Suppose we ran a model that had a very complex trace that takes a long time to compile, but only ran it for one iteration. Clearly, tracing is not worth it in this case; we would have been better off directly executing the model with no optimization. On the other hand, MH it typically run for millions of iterations, suggesting that the overhead is worthwhile. Where does the tradeoff lie in concrete terms? If we increase the number of possible traces, so each is used less, how much does performance decrease?

In order to obtain a complete picture of performance, we will thus evaluate the effect of tracing and incrementalization along two dimensions: number of random variables, and amount of structural change. We do so using the Ising model, a probability distribution originating from statistical physics which is simple yet provides challenging inference problems.

7.1 Evaluation on Ising model

In statistical physics, the Ising model depicts the magnetic character of a metal as the up $(+1)$ or down (-1) spin of discrete chunks within it, called *domains* (alternatively, *sites*). Each domain exerts influence on the spin of its neighbors within the metal, encouraging them to align, with the influence growing in intensity as the temperature of the metal decreases. We may represent a one-dimensional version of this model as a list of binary random variables, with factor functions modeling interactions between neighboring variables. We also consider extension of the Ising model to the open-universe case, where the number of domains in the metal is not known in advance. In this setting, the variation in the number of sites is represented by an integer random variable.

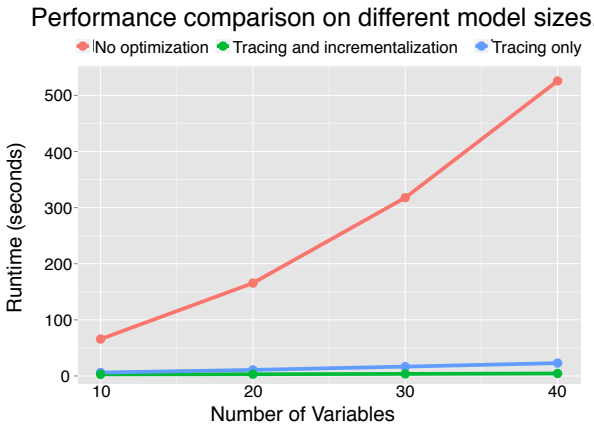

Figure 3. Performance of the algorithm on a fixed-size Ising model, as a function of the number of random variables, comparing different optimization settings.

For each model, we compared the performance of an unoptimized Church implementation (based on [Wingate et al. 2011-1]) versus tracing versus tracing with incrementalization. To evaluate the impact of running the tracer, we also ran each model for 100k and 1 million iterations, separately. We now describe each experiment in detail.

Experiment 1. Model size. We first consider the size of the model in terms of the number of random variables, using the following program:

```
(letrec
 ([eq (lambda (x y) (factor (if (= x y) 0.0 (log 0.1))))]
  [xs (repeat NUM-VARIABLES (lambda () (xrp+init
       randint-scorer randint randint-prop 0 0 1)))]
  [constr (map (lambda (xy) (eq (car xy) (car (cdr xy))))
       (bigram xs))])
 xs)
```

This gives the "generative story" behind the Ising model; we first define `eq`, expressing the interaction between any two sites. The spin state of each site is then sampled through the anonymous function passed to `repeat`, creating a list of such samples `xs` whose length is the number of sites we consider—`NUM-VARIABLES` which is the independent variable of this experiment.

We ran the above model with `NUM-VARIABLES` set to 10, 20, 30, and 40, for 1 million iterations, using no optimizations (A), tracing only (B), tracing plus incrementalization (C), and a hancoded implementation (D) where we introduced incrementalization manually, specialized to this particular model. The resulting runtimes are (also see Fig. 3):

Num. variables	A	B	C	D
10	65.7	6.14	2.83	1.67
20	165.48	10.78	3.12	1.69
30	317.66	16.70	3.69	1.67
40	525.73	22.94	4.42	1.67

We see that tracing by itself gave a large speedup over no optimizations. Tracing with incrementalization, further, gives greatly improved performance compared with just tracing.

Performance with incrementalization is also much less sensitive to the number of variables; the runtime for tracing with incremen-

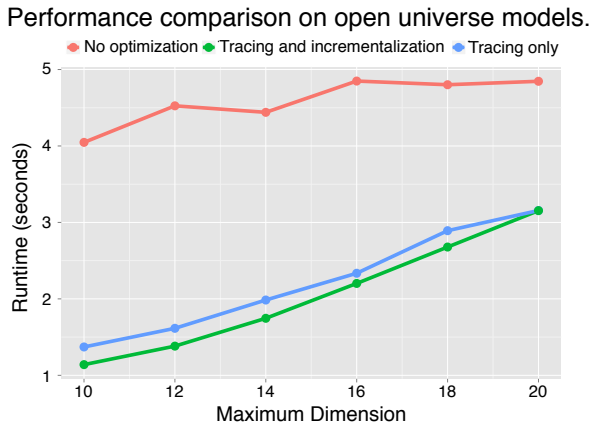

Figure 4. Performance of the open-universe Ising model (with LOW=8) as a function of HIGH, comparing different optimization settings.

talization starts out around 3 seconds and increases only somewhat (3.16 to 4.42 seconds going from 20 to 40 variables). In contrast, with tracing only, doubling the number of variables roughly doubled the running time (10.78 to 22.94 seconds going from 20 to 40 variables). This shows that slicing is able to exploit the independencies of this model appropriately.

Our compilation technique comes within an order of magnitude of a hand-coded and optimized implementation. This is fast enough to make the general-purpose implementation a compelling alternative to special-case implementation.

Experiment 2. Amount of structure change. We next evaluate the impact of structure change on our technique. Unlike the previous experiment (which had no structural variables) compilation cannot be performed at the outset, but must happen when a new setting of the structural variables is encountered. Consider the following modification to the model in Experiment 1, where we replace the `NUM-VARIABLES` variable by a sample from a uniform distribution over integers from `LOW` to `HIGH`, inclusive:

```
(letrec
 ([eq (lambda (x y) (factor (if (= x y) 0.0 (log 0.1))))]
  [n (randint LOW HIGH)]
  [xs (repeat n (lambda () (xrp+init randint-scorer
       randint randint-prop 0 0 1)))]
  [constr (map (lambda (xy) (eq (car xy) (car (cdr xy))))
        (bigram xs))])
 xs)
```

This captures an Ising model with uncertainty about the number of domains. We set `LOW` = 8 and vary `HIGH` in our experiments from `HIGH` =10 to `HIGH`=20 in increments of two. We ran 100K iterations for each of three levels of optimization: no optimizations, tracing only, tracing and incrementalization.

The results are summarized in Figure 4. We see a significant (order of magnitude) speedup due to tracing, and a small additional gain due to incrementalization. For longer runs and larger models we expect this small gain from incrementalization to magnify.

What causes the performance difference? Tracing clearly improved the performance of all examples as a function of model complexity, structure variation, and presence of independent substructures. Incrementalization further improved performance, but the improvement lessened with more structure change or less independent structure in the program. This raises further questions. How much time is spent creating/deleting random variables and executing control flow for the un-optimized implementation? How much time is spent generating versus compiling versus running traces for the trace-based implementations? To answer this, we will look at where each implementation spends its time.

We instrumented the code to perform profiling of functions related to evaluating the probabilistic program, keeping track of random variables, and generating/compiling traces. We re-ran some cases from Experiment 1 and 2, concentrating on those where the various implementations show performance differences. See Figure 5 for a visual breakdown of performance of the Ising models according to these functions. Note that profiling results in overhead itself, but lets us observe how much time is spent running traces versus compiling them. We test tracing only (Variation B) versus tracing with incrementalization (Variation C). For Experiment 1, we only ran the profiling test on 10 and 40 variables. For Experiment 2, we ran the most extreme cases—5K versus 100K iterations, and 9 to 11 versus 3 to 17 variables.

We see that for models with no structure change, it is very profitable to generate the traces: a neglible amount of time is spent performing tracing and retrieval for the profiled Experiment 1 runs. However, for the models from Experiment 2, which do have structure change, we see that a significant amount of time is spent compiling, caching, and retrieving traces. Yet, as we increase the number of iterations, it becomes worthwhile to generate traces; we spend a smaller portion of our time compiling traces and dealing with structure change.

The effect of incrementalization is made clearer by the performance profiles—we see that with incrementalization, the time spent running the kernel for the static Ising model is the same for the 40-variable model as it is for the 10-variable one, but without incrementalization, the runtime increases with the number of variables. The slightly higher runtime with incrementalization was thus due entirely to the time spent compiling the initial trace.

7.2 Comparisons with JAGS

We explored performance in depth for the Ising model; what about more complex models such as those targeted by existing statistical computation packages? We give preliminary performance results for comparison to one such system: Just Another Gibbs Sampler (JAGS). Unlike with Church and stochastic lambda calculus, JAGS cannot express models with structure change. Yet this restriction in expressivity means that it is easier to make general purpose inference tools (and indeed these tools are mature and in wide use). We will test our technique against JAGS using the following examples, taken from the JAGS manual:

1. **Linear regression.** The goal of linear regression is to infer the slope and intercept of a line that best fits a given set of x, y points. Linear regression is a common application of probabilistic inference. Cast as a generative model, it involves first *sampling* the slope and intercept, then computing the likelihood as a product of *factors*, one for each data point. This model contains very little independent structure and few random choices.
2. **Rat growth model.** This example infers the growth functions for the weight of several rats, whose weights were measured over the course of five weeks. We infer the intercept and slope, `alpharat, betarat` respectively, of each rat, assuming that they are taken from relatively uninformative priors `alphac, betac`. This model is equivalent to several linear regressions with tied parameters. Consequently, there are many random choices and some independent structure that our compiler can exploit.

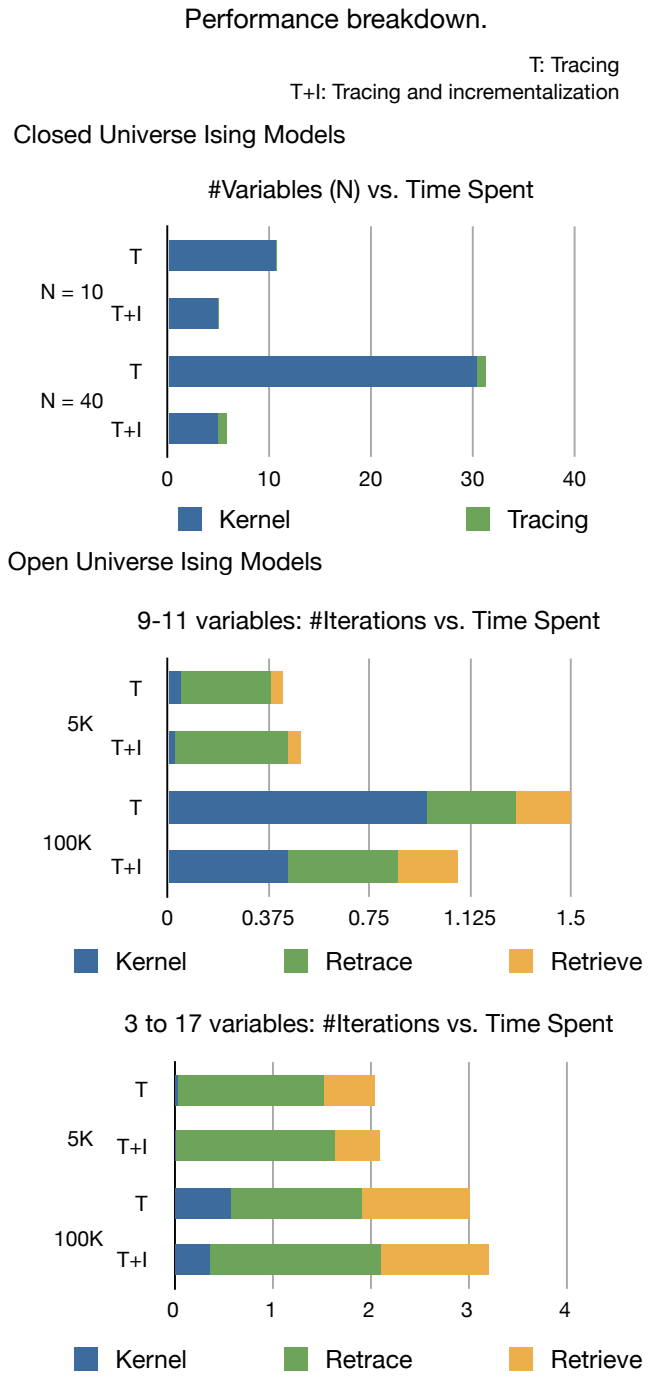

Figure 5. Amount of time spent (in seconds) tracing versus generating / compiling traces, for selected test cases. Comparing tracing-only (B bars) with tracing and incrementalization (C bars).

We ran both models for 1 million iterations in our system and in JAGS. Performance results were:

	Linear Regression	Rat Growth
Unoptimized	28.13 s	1514.37 s
Tracing + Incr.	10.10 s	25.82 s
JAGS	4.12 s	33.47 s

We see that the two systems are comparable in speed overall. JAGS is faster on linear regression, but is slower on the rat growth model. Thus our JIT compilation techniques makes inference for arbitrary probabilistic programs competitive with inference for specialized languages like JAGS—sometimes even faster. Further evaluation is needed to compare the statistical efficiency of these systems, since they don't use exactly the same inference technique, but this comparison is very encouraging.

8. Related Work

Probabilistic programming languages. Our system provides a much more efficient implementation of Metropolis-Hastings-based queries [Wingate et al. 2011-1] for the Church probabilistic programming language [Goodman et al. 2008]. Other inference algorithms for Church can be better for some classes of programs [Wingate et al. 2011-2, Stuhlmüller and Goodman 2012]. Our techniques could likely be used to accelerate these other Church algorithms, and also algorithms for a variety of other universal probabilistic programming languages (such as HANSEI [Kiselyov and Shan 2009] and Figaro [Pfeffer 2009]).

Other probabilistic programming languages take a different approach, using a simpler, non-Turing-complete language specialized for certain kinds of probabilistic computations. As we have discussed, BUGS and JAGS [Andrew 1994] [JAGS], which focus on simpler probabilistic models containing no structural choices. BLOG [Milch et al. 2007], short for Bayesian Logic, is another such a language often used to express probabilistic models where the identity and number of entities is unknown. While it cannot express all of the models that a universal language like Church or HANSEI can, the lower level of expressivity of BLOG often enables more accurate analyses and more aggressive optimizations. Even here, we believe the JIT techniques we have described could result in large speed improvements.

Just-in-time compilation. Just-in-time compilation has a long history [Aycock 2003]. Trace-based JIT compilation has received renewed interest. Much of the work focuses on applying such techniques to Javascript, a widely used language for specifying client-side scripts on web pages. In particular, by specializing the code in loops to the types actually used in them at runtime, speedups of an order of magnitude are possible [Gal et al 2009]. The most relevant work to ours is PyPy [Bolz et al. 2009], a system for performing trace-based JIT compilation generically. One writes an augmented interpreter for an object language of interest in RPython, and the resulting interpreter code is what undergoes JIT compilation. In particular, we use the method of allocation removal by partial evaluation [Bolz et al. 2011], an optimization that was developed for PyPy.

Incremental computation. We do not address incremental computation in the same generality as those who have worked on self-adjusting computation [Acar 2005]. Rather, we focus incrementalizing only our traces, which do not contain control flow and consist of a static sequence of function calls. We use reaching definitions analysis [Aho et al. 2006] to compute the set of statements affected by a proposal to a random choice. This is similar to the incrementalization of Scheme programs accomplished by combining a reaching definitions analysis with partial evaluation [Sundaresh 1991]. It may be that more general incrementalization techniques could provide additional improvements in our system.

9. Conclusion and Future Work

We have shown that applying techniques from JIT compilation can greatly speed up MCMC in probabilistic programs. This speedup can be of more than two orders of magnitude; the generated code runs even nearly as fast as hand-written code in certain situations. Moreover, the performance we achieve is comparable to that of

mature statistical packages where the language is much more restricted. This improvement is possible because MCMC is, on the whole, a repetitive computation with removable overhead. Our tracing scheme both removes the overhead (such as that caused by tracking random choices during program execution) and exploits decomposition of the acceptance probability into statistically independent pieces—we identify these pieces to avoid re-computing the entire ratio on each iteration.

9.1 Improvements

Perhaps the most immediate area for future work is in decreasing the time spent generating traces and compiling them. Overhead spent in this way increases with the number of possible structures in the model—i.e., the number of possible control flow paths. In general, this can be unbounded, but the traces from "nearby" structures are not independent; it may be possible to more efficiently account for structure changes by tracking shared structure between executions. In addition, by automatically inferring which variables are structural and which ones are not, models in our system would be easier to write.

In addition, the choice of target language in which to express traces has a large effect. We use Ikarus Scheme [Ghuloum 2008], a R5RS Scheme to x86 assembly compiler. Ikarus `eval` is able to compile expressions at runtime into x86 code. However, we could just as easily implement the tracing and slicing schemes discussed above for other target languages, such as Javascript code generators. This is possibly advantageous, as Javascript compiles quickly and our techniques could potentially be integrated with existing JIT compilers for Javascript.

Finally, the implementation of our tracer is syntax-directed; it is essentially an interpreter running in Ikarus Scheme. More efficient techniques for tracing, such as the virtual machine and higher order abstract syntax, may be useful extensions. In general, making the tracer and trace compilation more efficient would further increase the payoff of generating traces and compiling them.

9.2 Why optimize probabilistic programs?

The applications of probabilistic programming are not limited to machine learning. Probabilistic programming can also viably apply to other domains, such as computer graphics, where one can readily express complex procedural models with constraints [Yeh et al. 2012, Talton et al. 2011] as a query in a probabilistic programming language. We believe that making probabilistic programming languages faster, will encourage their use in a large and rich set of domains; this in turn will provide new applications of probabilistic programming.

More generally, however, there is a deep connection between probabilistic programming and program verification and analysis that warrants further exploration. In the former, we count the possible program paths satisfying a logical constraint, with paths weighted by their probabilities. In the latter, we would like to prove or disprove validity of such logical constraints, in addition to generating additional constraints, such as preconditions, that may be simpler or more useful. There is great potential for each discipline to inform the others. Probabilistic inference techniques such as MCMC have already proven to be of use in program verification [Gulwani and Jojic 2007]. Conversely, if we can generate constraints so that it is not necessary to explore all program paths to obtain a probability, it will result in faster inference algorithms. Once we start answering the question of computing probabilistic program queries faster, we also stumble upon questions about how to reason about general program behavior. With the wide variety of application areas of probabilistic programming, we will surely find new and useful answers to them.

References

[Acar 2005] U. Acar. Self-Adjusting Computation. Ph. D thesis (2005).

[Aho et al. 2006] A. Aho, M. S. Lam, R. Seth, and J. D. Ullman. Compilers: Principles, Techniques, and Tools (2nd edition). Prentice Hall, 2006.

[Andrew 1994] A. Thomas. BUGS: a Statistical Modeling Package. RTA/BCS Modular Languages Newsletter, Vol. 2, pp. 36–38. 1994.

[Aycock 2003] J. Aycock. A Brief History of Just-in-Time. ACM Computing Surveys 2003, Vol. 35, pp. 97–113.

[Barrett et al. 2010] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard—Version 2.0. In Gupta, A., Kroening, eds. SMT 2010.

[Bolz et al. 2009] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In proceedings of ICOOOLPS 2009, pp. 18–25.

[Bolz et al. 2011] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation Removal by Partial Evaluation in a Tracing JIT. In proceedings of PEPM 2011, pp. 43–52.

[Freer and Roy 2010] C. E. Freer and D. M. Roy. Posterior Distributions are Computable from Predictive Distributions. In proceedings of AISTATS 2011, pp. 233–240.

[Gal et al 2009] A. Gal, B. Eich, M. Shaver, D. Anderson, and D. Mandelin. Haghighat, Mohammad R. Kaplan, Blake. Hoare, Graydon. Zbarsky, Boris. Orendorff, Jason. Ruderman, Jesse. Smith, Edwin. Reitmaier, Rick. Bebenita, Michael. Chang, Mason. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In proceedings of PLDI 2009, pp 465–478.

[Ghuloum 2008] A. Ghulom. Ikarus Scheme User's Guide. https://launchpadlibrarian.net/18248997/ikarus-scheme-users-guide.pdf. 2008.

[Goodman et al. 2008] Goodman, Noah. Mansinghka, Vikash K. Roy, Daniel M. Bonawitz, Keith. Tenenbaum, Joshua B. Church: a Language for Generative Models. In proceedings of UAI 2008, pp. 220–229.

[Gulwani and Jojic 2007] S. Gulwani and N. Jojic. Program Verification as Probabilistic Inference. In proceedings of POPL 2007.

[Hastings 1970] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. Biometrika 57 (1), pp. 97–109. 1970.

[JAGS] Just Another Gibbs Sampler. http://mcmc-jags.sourceforge.net.

[Kiselyov and Shan 2009] O. Kiselyov and C. Shan. Embedded Probabilistic Programming. IFIP working conference on domain-specific languages. Walid Taha, editor. LNCS 5658, Springer, 2009, pp. 360–384.

[McCallum et al. 2007] A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In proceedings of NIPS 2009, pp. 1249–1257.

[Milch et al. 2007] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic Models with Unknown Objects. In proceedings of IJCAI 2005, pp. 1352–1359.

[Pfeffer 2009] A. Pfeffer. Figaro: An Object-Oriented Probabilistic Programming Language. Charles River Analytics Technical Report 2009.

[Ramsey and Pfeffer 2002] N. Ramsey and A. Pfeffer. Stochastic Lambda Calculus and Monads of Probability Distributions. In proceedings of POPL 2002, Vol. 37 No. 1, pp. 154–165.

[Stuhlmüller and Goodman 2012] A. Stuhlmüller and N. D. Goodman. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. In proceedings of the Second Statistical Relational AI workshop at UAI 2012.

[Sundaresh 1991] R. S. Sundaresh. Building Incremental Programs using Partial Evaluation. In proceedings of PEPM 1991, pp. 83–93.

[Talton et al. 2011] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun. Metropolis Procedural Modeling. ACM Transactions on Graphics 2011, Vol. 30 No. 2, pp. 11:1–11:14.

[Wingate et al. 2011-1] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight Implementations of Probabilistic Programming Languages

Via Transformational Compilation. In proceedings of UAI 2011, pp. 1–9.

[Wingate et al. 2011-2] D. Wingate, N. D. Goodman, A. Sthulmüller, and J. M. Siskind. Nonstandard Interpretations of Probabilistic Programs for Efficient Inference. In proceedings of NIPS 2011.

[Yeh et al. 2012] Y. -T. Yeh, L. Yang, M. Watson, N. D. Goodman, and P. Hanrahan. Synthesizing Open Worlds with Constraints using Locally Annealed Reversible Jump MCMC. ACM Transactions on Graphics 2012, Vol. 31 No. 4, pp. 56:1–56:11.