Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation

David Wingate
wingated@mit.edu
Brain and Cognitive Sciences
MIT

Andreas Stuhlmüller
ast@mit.edu
Brain and Cognitive Sciences
MIT

Noah D. Goodman
ngoodman@stanford.edu
Department of Psychology
Stanford

Abstract

We describe a general method of transforming arbitrary programming languages into probabilistic programming languages with straightforward MCMC inference engines. Random choices in the program are "named" with information about their position in an execution trace; these names are used in conjunction with a database holding values of random variables to implement MCMC inference in the space of execution traces. We encode naming information using lightweight source-to-source compilers. Our method enables us to reuse existing infrastructure (compilers, profilers, etc.) with minimal additional code, implying fast models with low development overhead. We illustrate the technique on two languages, one functional and one imperative: Bher, a compiled version of the Church language which eliminates interpretive overhead of the original MIT-Church implementation, and Stochastic Matlab, a new open-source language.

1 INTRODUCTION

Probabilistic programming languages simplify the development of probabilistic models by allowing programmers to specify a stochastic process using syntax that resembles modern programming languages. These languages allow programmers to freely mix deterministic and stochastic elements, resulting in tremendous modeling flexibility. The resulting programs define prior distributions: running the (unconditional) program forward many times results in a distribution over execution traces, with each trace generating a sample of data from the prior. The goal of inference in such programs is to reason about the posterior distribution over execution traces conditioned on a particular program output. Examples include BLOG [7], PRISM [15], Bayesian Logic Programs [5] Stochastic Logic Programs [9], Markov Logic [14], Independent Choice Logic [12], IBAL [11], Probabilistic Scheme [13], $\Lambda_\circ$ [10], and HANSEI [6].

We present a general technique for turning any programming language into a probabilistic programming language with an accompanying universal Markov chain Monte Carlo inference engine. Our method allows the full use of all language constructs permitted by the original (non-probabilistic) language by leveraging native compilers (or interpreters). The key technical idea is to give each random choice of a fixed program a unique "name" that depends on its position in a given execution trace. We then convert stochastic functions into deterministic ones which use these names to look up their return values in a database that stores the values of all random choices. By controlling the values in this database, execution traces of the program can be controlled, which allows us to construct the key operations needed for the Metropolis-Hastings algorithm.

Different techniques for naming random choices result in different MCMC dynamics. Our central contribution is to name random choices based on their structural position in the execution trace, which allows fine-grained sharing of random choice values between subsequent MCMC states. We describe how naming information can be provided by a side computation that is constructed by a source-to-source transformation at compile time: the original probabilistic program can be augmented with naming information to create a new program that makes the names available. The resulting new program can be executed at full speed, with only minimal overhead to track names and control the database, and no changes to the compiler or interpreter for the underlying language.

We illustrate our technique on two languages, one functional and one imperative. The first is named "Bher,"

Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15. Copyright 2011 by the authors. **Revision 3. Updated February 8, 2014.**

and is a compiler for the Church probabilistic programming language [4]. The current MIT-Church implementation uses a custom interpreter to reason about the distribution of execution traces, but since our method does not incur interpretive overhead, the resulting programs can run orders of magnitude faster. We also illustrate the technique on an imperative language by creating Stochastic Matlab, an entirely new language.

2 OVERVIEW OF THE METHOD

We begin by outlining our setup. We define an unconditioned probabilistic program to be a parameterless function f with an arbitrary mix of stochastic and deterministic elements (hereafter, we will use the term function and program interchangeably). The function f may be written in any language, but our running example will be Matlab. We allow the function to be arbitrarily complex inside, using any additional functions, recursion, language constructs or external libraries it wishes. The only constraint is that the function must be self-contained, with no external side-effects which would impact the execution of the function from one run to another.

The stochastic elements of f must come from a set of known, fixed *elementary random primitives*, or ERPs. Complex distributions are constructed compositionally, using ERPs as building blocks. In Matlab, ERPs may be functions such as `rand` (sample uniformly from [0,1]) or `randn` (sample from a standard normal). Higher-order random primitives, such as nonparametric distributions, may also be defined, but must be fixed ahead of time. Formally, let $\mathcal{T}$ be the set of ERP types. We assume that each type $t \in \mathcal{T}$ is a parametric family of distributions $p_t(x|\theta_t)$, where θ_t are the parameters of the distribution.

As f is executed, it encounters a series of ERPs. Algorithm 1 shows an example of a simple f written in Matlab. Here, there are three syntactic ERPs: `rand`, `randn`, and `gammarnd`. During execution, depending on the return value of each call to `rand`, different paths will be taken through the program, and different ERPs will be encountered. We call this path an *execution trace*. A total of 2000 random choices will be made when executing this f.

Let $f_{k|x_1,\cdots,x_{k-1}}$ be the k'th ERP encountered while executing f, and let x_k be the value it returns. Note that the parameters passed to the k'th ERP may change depending on previous x_k's (indeed, its type may also change, as well as the total number of ERPs). We denote by x all of the random choices which are made by f, meaning that f defines the probability distribution $p(x)$. In our example, $x \in \mathbb{R}^{2000}$. The probability of x is therefore the product of the probability of all of the ERP choices made:

$$p(x) = \prod_{k=1}^{K} p_{t_k}(x_k|\theta_{t_k}, x_1, \cdots, x_{k-1})$$

Algorithm 1 A simple Stochastic Matlab program representing a Gaussian-Gamma mixture model.

```
for i=1:1000
    if ( rand > 0.5 )
        X(i) = randn;
    else
        X(i) = gammarnd;
    end;
end;
```

again noting explicitly that types and parameters may depend arbitrarily on previous random choices. To simplify notation, we will omit the conditioning on the values of previous ERPs, but again wish to emphasize that these dependencies are critical and cannot be ignored. By f_k, it should therefore be understood that we mean $f_{k|x_1,\cdots,x_{k-1}}$, and by $p_{t_k}(x_k|\theta_{t_k})$ we mean $p_{t_k}(x_k|\theta_{t_k}, x_1, \cdots, x_{k-1})$.

Generative functions as described above are, of course, easy to write. A much harder problem, and our goal in this paper, is to reason about the posterior conditional distribution $p(x_{\setminus c}|x_c)$, where we define x_c to be a subset of random choices which we condition on and $x_{\setminus c}$ to be the remaining random choices. For example, we may condition our example program on the values of the X(i)'s, and reason about the sequence of `rand`'s most likely to generate the X(i)'s. Rejection sampling is one possible inference method: we may run f forward until x_c is generated, somehow recording $x_{\setminus c}$. Of course, such an approach is intractable for all but the simplest models.

We propose a different approach, based on the following intuition. Consider repeatedly running an unconditioned f. Every time an f_k is encountered, the program samples an x_k and continues execution. Importantly, if two runs of f sample exactly the same values, their traces will be the same. This suggests the following procedure for controlling the execution of f:

1. Give each f_k in every possible execution trace a "name." This name does not have to be unique, and can depend on previous x_k's or other information about program state.
2. Rewrite the source code of f to generate f', replacing random functions f_k with deterministic functions f'_k. When encountered in the execution trace, these functions f'_k deterministically use their name to look up a current value x_k in a database and return it (and if no value exists, they sample $x_k \sim p_{t_k}(.|\theta_{t_k})$., store it in the database, and then return it). Behind the scenes, these functions also accumulate the likelihood of each random choice.

Thus, the execution trace of f' can be controlled by manipulating the values in the database. This makes MCMC

Algorithm 2 An MCMC trace sampler.

Initialize: $[\,\mathbb{D}, ll, _, _\,]$ = trace_update($\emptyset$)
Repeat forever:
  Select a random f_k via its name n
  Look up its current value $(t, x, l, \theta_{db}) = \mathbb{D}(n)$.
  Propose a new value $x' \sim \mathcal{K}_t(.|x, \theta_{db})$
  Compute $F = \log \mathcal{K}_t(x'|x, \theta_{db})$
  Compute $R = \log \mathcal{K}_t(x|x', \theta_{db})$
  Compute $l' = \log p_t(x'|\theta_{db})$
  Let $\mathbb{D}' = \mathbb{D}$
  Set $\mathbb{D}'(n) = (t, x', l', \theta_{db})$
  $[\,\mathbb{D}', ll', ll_{\text{fresh}}, ll_{\text{stale}}\,]$ = trace_update($\mathbb{D}'$)
  $\alpha = ll' - ll + R - F + \log|\mathbb{D}| - \log|\mathbb{D}'| + ll_{\text{stale}} - ll_{\text{fresh}}$
  if (log(rand) $< \alpha$)
    // accept
    $\mathbb{D} = \mathbb{D}'$
    $ll = ll'$
  else
    // reject; discard $\mathbb{D}'$

inference possible by enabling proposals, scoring, and the ability to accept or reject proposals. We make a proposal to an execution trace by picking an x_k in the database, modifying it according to a proposal distribution, and then re-executing f' and computing the likelihood of the new trace. As f' is executed, some random choices may be reused, new randomness may be triggered (which is sampled from the prior), and randomness that was previously used may no longer be needed. In our example, we can make a proposal to, say, the 500th `rand`. This will change the execution trace, switching between the Gaussian branch and the gamma branch.

We now address the details. Section 2.1 discusses the overall MCMC algorithm, Section 2.2 discusses how random variables are named, and Section 2.3 discusses conditional inference.

2.1 MCMC IN TRACE SPACE

We now describe our MCMC algorithm. Here, we assume that some naming scheme has been defined, and that a mechanism for computing those names has been implemented; we will discuss how to do this in the next section.

Let a database $\mathbb{D}$ be defined as a mapping $\mathbb{N} \to \mathcal{T} \times X \times \mathbb{L} \times \theta$, where $\mathbb{N}$ is the name of a random choice, $\mathcal{T}$ is its ERP type, X is its value, θ are ERP parameters, and $\mathbb{L}$ is this random value's likelihood. We allow missing entries.

We can control the execution of f' by controlling the values in $\mathbb{D}$. When f' encounters an f'_k, it computes its name $n \in \mathbb{N}$, its parameters θ_c (where 'c' is a mnemonic for "current"), and its current type $t_c \in \mathcal{T}$. It then looks up $(t, x, l, \theta_{db}) = \mathbb{D}(n)$. If a value is found and the types

Algorithm 3 Function `trace_update( 𝔻 )`

Initialize ll, ll_{fresh}, and ll_{stale} to 0
Execute f**:**
For all random choices k**:**
  Run computation until choice k,
  determining n, t_c, θ_c for k.
  Look up $(t, x, l, \theta_{db}) = \mathbb{D}(n)$
  if a value is found in the database and $t = t_c$
    if parameters match (ie, $\theta_c == \theta_{db}$)
      $ll = ll + l$
    else
      // rescore ERP with new parameters
      Compute $l = \log p_{t_c}(x|\theta_c)$
      Store $\mathbb{D}(n) = (t_c, x, l, \theta_c)$
      $ll = ll + l$
  else
    // sample new randomness
    Sample $x \sim p_t(.|\theta_c)$
    Compute $l = \log p_t(x|\theta_c)$
    Store $\mathbb{D}(n) = (t_c, x, l, \theta_c)$
    $ll = ll + l$
    $ll_{\text{fresh}} = ll_{\text{fresh}} + l$
  Set return value of f_k to x
For all random choices k **in** $\mathbb{D}$**:**
  if k has not been active in this update:
    Look up $(t, x, l, \theta_{db}) = \mathbb{D}(k)$
    $ll_{\text{stale}} = ll_{\text{stale}} + \log p_t(x|\theta_{db})$
    Remove k from $\mathbb{D}$
return $[\,\mathbb{D}, ll, ll_{\text{fresh}}, ll_{\text{stale}}\,]$

match, we set the return value of f_k to be x. Otherwise, the value is sampled from the appropriate ERP $x \sim p_{t_c}(.|\theta_c)$, its likelihood is computed, and the corresponding entry in the database is updated. We accumulate the log-probability of new randomness, ll_{fresh}, and the log-probability of randomness that is no longer used, ll_{stale}. This is formalized in the `trace_update` procedure, defined in Algorithm 3.

We use `trace_update` to define our overall MCMC algorithm, shown in Algorithm 2. Given a current trace x and score $p(x)$, we proceed by reconsidering one random choice x_k. We equip each ERP type with a proposal kernel $\mathcal{K}_t(x'|x, \theta)$, which we use to generate proposals to x_k. After a proposal, we call `trace_update` to generate a new trace x', and compute its likelihood $p(x')$ and the probabilities of new and stale randomness, p_{fresh} and p_{stale}. This gives us an overall score, which is used as an MH accept ratio

$$\alpha = \min\left\{1, \frac{p(x')\,\mathcal{K}_t(x|x', \theta)\,|\mathbb{D}|\,p_{\text{stale}}}{p(x)\,\mathcal{K}_t(x'|x, \theta)\,|\mathbb{D}'|\,p_{\text{fresh}}}\right\}$$

There are a few subtleties to the implementation. Even if an x for an f'_k is found in $\mathbb{D}$, we must compare the pa-

Algorithm 4 An illustration how the number and type of random choices can change.

```
1: m = poissrnd();
2: for i=1:m
3:     X(i) = gammarnd();
4: end;
5: for i=m+1:2*m
6:     X(i) = randn;
7: end;
```

rameters θ_c and θ_{db} to make sure they are equal; if they are not, we may still be able to re-use x, but must rescore it under the new parameters θ_c. We also remove unused ("stale") random choices from the database to ensure that subsequent new randomness is sampled from the prior, and allows us to avoid reversible jump corrections. Finally, note that many of the probabilities in α cancel when proposals are made from the local prior.

2.2 NAMING RANDOM VARIABLES

What properties do we desire in a naming scheme? We build around a simple idea: when we propose a new x'_k and update the trace, we wish to reuse as many x_k's as possible. This will give us a new execution trace "close" to the original, which is important to ensure a high acceptance rate in our MCMC sampler (recall that if an f'_k cannot reuse a value from $\mathbb{D}$, one is sampled from the prior, which is unlikely to be good).

How might different naming schemes affect reuse of randomness? Algorithm 4 shows a simple program f which we will use to illustrate different approaches. As f executes, it first encounters a Poisson, and then a sequence of gamma and Gaussian variables. One might be tempted to name random variables sequentially, according to the order in which they are encountered during execution—that is, use k itself as the name. Figure 1 (top) illustrates why this is a bad idea. Two traces are shown schematically, illustrating the f'_k's encountered. Suppose that in trace #1, the value of `m` is 3. A proposal is made, changing `m` to 2, yielding trace #2. If variables are named sequentially, then x_1 and x_2 can be reused without problem, but when f'_3 looks itself up in $\mathbb{D}$, it discovers that its type has changed, and must sample x_3 from the prior. The rest of the sequence is now "misaligned," and must probably be discarded. More sophisticated versions of this problem complicate this naming scheme to the point of unworkability, especially with complex models.

What is an alternative? Examining Alg. 4, we see that would like to name variables according to the way they are used at a more semantic level. For example, we might want to name a variable "the third gamma variable in the loop on lines 2-4."

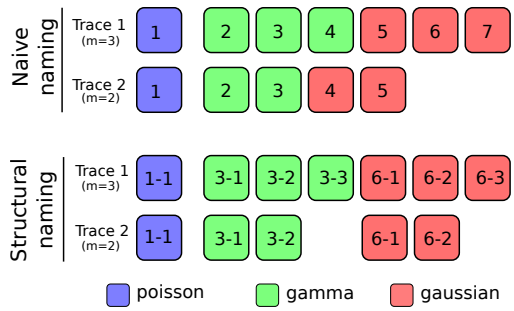

Figure 1: An illustration of two naming schemes. Variables are boxes with names inside. Top: the problem with naive variable naming. Notice the type mismatch at time 4 as well as the general sequence misalignment. Bottom: structural naming yields the desired behavior, reusing the maximum number of random choices.

Our key technical contribution formalizes this intuition. We name random choices according to their *structural position* in the execution trace, which we define in a way roughly analogous to a stack address: a variable's name is defined as list of the functions, their line numbers, and their loop indices that precede it in the call stack. Thus, in Alg. 4, variables are named "1" (the initial Poisson), "3-1,", "3-2," ... (the gamma variables sampled on line 3), and "6-1,", "6-2," ... (the Gaussian variables sampled on line 6). Figure 1(bottom) illustrates this on Alg. 4: we see that the maximum number of random variables are reused.

We now discuss naming schemes for imperative and functional languages.

Imperative. For an imperative language, such as Matlab, we define the structural position with an abstract stack trace that is augmented with a combination of function identifiers, line numbers, and loop iteration numbers. The formal specification is inductive, as shown in Figure 2. In an interpreted imperative language, such as Python, Matlab or Ruby, this naming information could potentially be acquired via dynamic stack examination, assuming suitable libraries exist. Alternatively, this information can be provided by augmenting the source code for f, as we shortly discuss.

Functional. For a functional language, such as Lisp, we do not need to worry about loops or line numbers: everything is a function, so we may define the structural position with a stack trace that is augmented with function identifiers. The specification is shown in Figure 3.

There is no guarantee that this naming scheme is optimal in any sense, but it does capture our desiderata, is simple and fast, and incurs minimal overhead.

$\mathcal{A}^{top}[\![E]\!] = ((\texttt{lambda (addr)}\ \mathcal{A}[\![E]\!])\ \texttt{'(top)})$
$\mathcal{A}[\![(\texttt{lambda}\ (I_{i=1}^n)\ E_{body})]\!] = (\texttt{lambda (addr .}\ I_{i=1}^n)\ \mathcal{A}[\![E_{body}]\!])$
$\mathcal{A}[\![(\texttt{mem}\ E)]\!] = ((\texttt{lambda (maddr f) (lambda (addr . args) (apply f (cons args maddr) args))) addr}\ \mathcal{A}[\![E]\!])$
$\mathcal{A}[\![(\texttt{begin}\ E_{i=1}^n)]\!] = (\texttt{begin}\ \mathcal{A}[\![E_i]\!]_{i=1}^n)$
$\mathcal{A}[\![(\texttt{letrec}\ ((I_i\ E_i)_{i=1}^n)\ E_{body})]\!] = (\texttt{letrec}\ ((I_i\ \mathcal{A}[\![E_i]\!])_{i=1}^n)\ \mathcal{A}[\![E_{body}]\!])$
$\mathcal{A}[\![(\texttt{if}\ E_t\ E_c\ E_a)]\!] = (\texttt{if}\ \mathcal{A}[\![E_t]\!]\ \mathcal{A}[\![E_c]\!]\ \mathcal{A}[\![E_a]\!])$
$\mathcal{A}[\![(\texttt{define}\ I\ E)]\!] = (\texttt{define}\ I\ \mathcal{A}[\![E]\!])$
$\mathcal{A}[\![(\texttt{quote}\ E)]\!] = (\texttt{quote}\ E)$
$\mathcal{A}[\![(E_{op}\ E_{i=1}^n)]\!] = (\mathcal{A}[\![E_{op}]\!]\ (\texttt{cons}\ \texttt{'}\mathcal{S}\ \texttt{addr})\ \mathcal{A}[\![E_i]\!]_{i=1}^n)$ with $\mathcal{S}$ being a globally unique symbol.
$\mathcal{A}[\![E]\!] = E$, otherwise.

Figure 3: The naming specification for Bher is given by a syntactic transform $\mathcal{A}^{top}$.

- Begin executing f with empty function, line, and loop stacks.
- When entering a new function,
  - push a unique function id on the function stack.
  - push a 0 on the line stack.
- When moving to a new line, increment the last value on the line stack.
- When starting a loop, push a 0 on the loop stack.
- When iterating through a loop, increment the last value on the loop stack.
- When exiting a loop, pop the loop stack.
- When exiting a function, pop the function stack and the line stack.

Figure 2: The imperative naming specification. The name of an f_k is the state of the three stacks when f_k is encountered.

$\mathcal{L}[\![(\texttt{lambda}\ (I_{i=1}^n)\ E_{body})]\!] = (\texttt{lambda}\ (I_{i=1}^n)\ \mathcal{L}[\![E_{body}]\!])$
$\mathcal{L}[\![(\texttt{letrec}\ ((I_i\ E_i)_{i=1}^n)\ E)]\!] = (\texttt{letrec}\ ((I_i\ \mathcal{D}[\![E_i]\!])_{i=1}^n)\ \mathcal{L}[\![E]\!])$
$\mathcal{L}[\![(\texttt{begin}\ E_{i=1}^n)]\!] = (\texttt{begin}\ (\texttt{force}\ \mathcal{L}[\![E_i]\!])_{i=1}^{n-1}\ \mathcal{L}[\![E_n]\!])$
$\mathcal{L}[\![(\texttt{mem}\ E)]\!] = (\texttt{mem}\ \mathcal{L}[\![E]\!])$
$\mathcal{L}[\![(\texttt{quote}\ E)]\!] = (\texttt{quote}\ E)$
$\mathcal{L}[\![(\texttt{if}\ E_t\ E_c\ E_a)]\!] = (\texttt{if}\ (\texttt{force}\ \mathcal{L}[\![E_t]\!])\ \mathcal{D}[\![E_c]\!]\ \mathcal{D}[\![E_a]\!])$
$\mathcal{L}[\![(E_{op}\ E_{i=1}^n)]\!] = ((\texttt{force}\ \mathcal{L}[\![E_{op}]\!])\ \mathcal{D}[\![E_i]\!]_{i=1}^n)$
$\mathcal{L}[\![E]\!] = E$, otherwise.
$\mathcal{D}[\![(\texttt{lambda}\ I\ E)]\!] = \mathcal{L}[\![(\texttt{lambda}\ I\ E)]\!]$
$\mathcal{D}[\![(\texttt{mem (lambda}\ I\ E))]\!] = \mathcal{L}[\![(\texttt{mem (lambda}\ I\ E))]\!]$
$\mathcal{D}[\![E]\!] = (\texttt{list 'delayed (mem (lambda () }\mathcal{L}[\![E]\!]))$, otherwise.

```
(define (force address val)
  (if (and (pair? val) (eq? (car val) 'delayed))
    (force address ((cadr val) address)))
    val)
```

Figure 4: Laziness transform $\mathcal{L}$ and definition of the function `force` referenced in the transform.

2.3 CONDITIONAL INFERENCE

In principle, a program can be conditioned on any expression that evaluates to a Boolean value, not just on a fixed value assignment for a subset of the random variables of the program. The meaning of such a conditioned program is defined by rejection sampling: Sample an execution trace from the unconditioned program. If the value of the conditioning expression is `true`, return this trace. Otherwise, retry.

In practice, a MCMC implementation that allows general conditions requires that for any valid condition, we can find an initial assignment of random variables to values that satisfies this condition. If conditioning is not restricted to random variables (Stochastic Matlab), then this is a search problem that needs to be solved. Possible approaches include constraint propagation (MIT-Church), annealing (Bher), and rejection sampling (Bher). We leave the analysis of this problem to future research.

3 IMPLEMENTATION AND EXPERIMENTS

We now present two implementations of our technique. Bher is a compiled implementation of Church, and Stochastic Matlab is a new, open-source language. Both illustrate the virtues of our approach: small code bases, high speed, and language flexibility.

3.1 EXAMPLE: BHER

Bher is an implementation of the Church probabilistic programming language that compiles Church code to native Scheme code. Like MIT-Church, the original implementation of the Church language, Bher is written in R6RS Scheme. However, MIT-Church is an *interpreter*—not based on the lightweight implementation principle—and thus provides a useful point of comparison.

In the following, we first discuss the addressing transform that names random choices in Bher, then compare performance between Bher and MIT-Church on Hidden Markov Models and Latent Dirichlet Allocation.

Algorithm 5 A Church program that samples from a geometric distribution.

```
(begin
  (define geometric
    (lambda (p)
      (if (flip p)
          1
          (+ 1 (geometric p)))))
  (geometric .7))
```

3.1.1 Syntactic Transformations

The syntactic transform that provides addresses is shown in Figure 3. The address that is built at runtime consists of a list of symbols. The transform is designed to change the program code such that a new symbol is added to the front of this list whenever the stack is extended.

The transform has two central parts: First, change each function definition such that the function takes another argument, the address. Second, change each function application such that (1) the current address is extended with a symbol that uniquely identifies this (syntactic) application within the program and (2) pass on this modified address as an argument when the function is applied.

On the top level of the program, the address variable `addr` is initialized to `'(top)`. Except for memoization (`mem`) and `quote`, all remaining syntax cases are handled by applying the addressing transform to all proper subexpressions. Primitive functions like `+` and `cons` need to be redefined to take (and ignore) the address argument, e.g. to `(lambda (addr . args) (apply prim args))`. In Bher, all globally free variables are treated as primitives and redefined in this way.

In deterministic programs, memoization is an optimization technique that does not affect semantics: When a memoized function returns, it stores the return value for its current set of arguments in a cache. When the function is called a second time with the same arguments, the value is not computed again, but instead the value found in the cache is returned.

In stochastic programs, we can distinguish `mem` as a semantic construct from `mem` as an optimization technique [4]. The distribution defined by `(eq? (flip) (flip))` is very different from the distribution defined by `(let ([mf (mem flip)]) (eq? (mf) (mf)))`. The former is true with probability .5, false with probability .5; the latter is true with probability 1.

When addresses are available, semantic memoization is a simple syntactic transform. In a program that has been transformed to provide addresses, random variables get their values by looking up the value that is stored in the global database at their address. Thus, to enforce that two random choices always have the same value, we make sure that the address they receive is the same. To achieve this, the `mem` case of the addressing transform builds a function that captures within its closure the address of the `mem` creation and uses this address (extended by any function arguments) instead of the address provided to the function when it is applied.

Algorithm 6 Transformed version of the Bher program shown in Algorithm 5. 'a1 to 'a4 are the names generated for function applications.

```
((lambda (addr)
   (begin
     (define geometric
       (lambda (addr p)
         (if (flip (cons 'a1 addr) p)
             1
             (+ (cons 'a2 addr)
                1
                (geometric (cons 'a3 addr) p)))))
     (geometric (cons 'a4 addr) 0.7)))
 '(top))
```

The Church program shown in Algorithm 5 and its transformed version (Algorithm 6) illustrate the address computation that is induced by the syntactic transform. The program defines a simple geometric distribution with a potentially unbounded number of random choices, each of which gets a unique address. When we propose to change the value of the `flip` that controls the recursion depth, we want to reuse as much of the existing trace as possible. An example of such a trace is shown in Figure 5. Note that apply nodes (shown in green) are the only place where addresses change and that nodes that refer to the same *syntactic* place (e.g. the two blue `(flip p)` nodes) have different addresses when they occur in different places in the trace. A change to the return value of such a `flip` does not affect the addresses of the choices "above" it. Thus, any randomness stored for these choices persists across the change.

A major virtue of source-to-source transformations is simplicity and compositionality. To illustrate this point, we now sketch a complementary transformation, eager style to lazy style, that can be used to avoid unnecessary computation. As shown in Figure 4, we delay expressions that must not necessarily get evaluated—for example, arguments to compound functions—and force expressions that are always evaluated—operators, for example. To delay an expression `E`, we wrap it in `(mem (lambda () E))`. Thus, we ensure that it always returns the same result when evaluated multiple times. To force a value, we check whether it is delayed and if it is, we simply apply the memoized function stored in the value and recurse.

This difference in simplicity manifests itself in the code base of the two projects. While the MIT-Church imple-

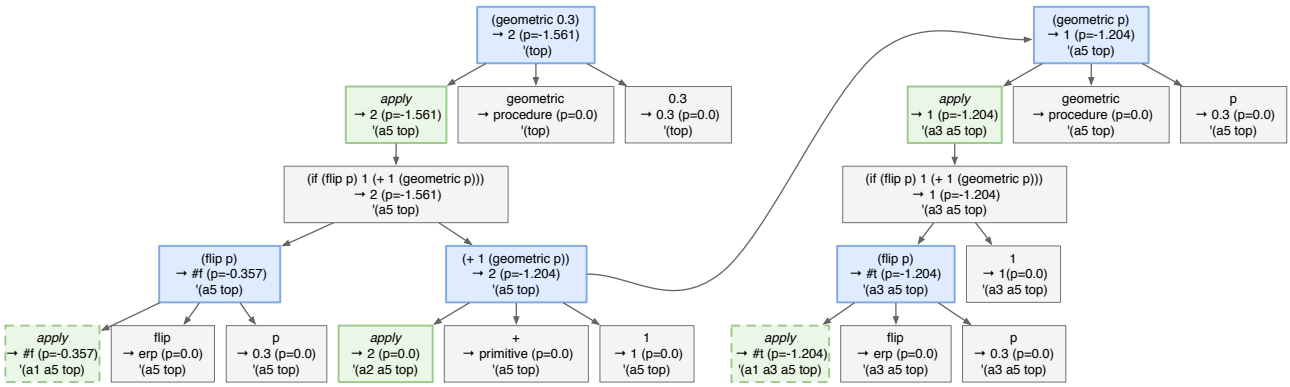

Figure 5: Example of an execution trace for the geometric sampler defined by Algorithm 5. Each node is annotated with its expression, value, log probability, and address. Applications (blue nodes) have subtraces for operator, operands, and for the application of operator to operands (*apply*, green nodes; dashed border indicates a random choice).

mentation has about 11,119 lines of Scheme code, Bher consists of only 1,658 lines of code.

3.1.2 Experiments

We have tested inference performance in MIT-Church and Bher on Hidden Markov Models and Latent Dirichlet Allocation (Figure 6). In both cases, we conditioned on an observation sampled from the model. For each HMM, we generated a sentence of length 15. For LDA, we used 10 topics, 10 words per document, and 30 possible words, varying the number of observed documents. All experiments use the optimizing compiler Ikarus [3].

On the HMM model, Bher outperforms the MIT-Church interpreter. On the LDA model, Bher shows better performance at smaller problem sizes whereas MIT-Church scales better to larger instances. This difference in scaling is due to (1) reuse of cached computation in `mem` and (2) computational short-circuiting in MIT-Church. In the process of updating a trace after a proposal has been made, MIT-Church can detect that the information flowing into a function application has not changed; in this case, it does not recompute the return value. Since `mem` is transformed away by the compiler, perhaps there is a source transformation that results in similar reuse of computation. This is an area of future research.

3.2 EXAMPLE: STOCHASTIC MATLAB

We now illustrate our technique to create a new, imperative probabilistic programming language called "Stochastic Matlab," which can take advantage of Matlab's rich native libraries and toolboxes, and its parallelization, profiling and debugging support.

To rewrite f, we used the imperative ERP naming specification as outlined in Fig. 2. We implemented the rewriter using a parser developed with code from the open-source

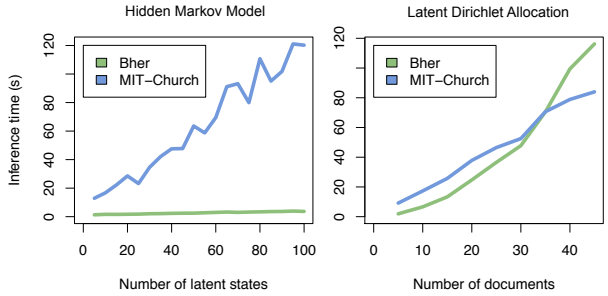

Figure 6: Inference performance for the HMM (10,000 samples) and LDA model (1,000 samples).

Octave project [2]. ERP names were rewritten to deterministic functions using a MEX-based database. Remaining code was implemented in pure Matlab.

We now present a case study that relies on a deterministic GPU-based MEX function. The point is twofold: first, to show how easy it is to mix arbitrary deterministic elements with stochastic ones—reusing complex external libraries—and second, to show how easy it can be to create complex models with a few simple commands.

Our example is a small (but real) application of inverse geological modeling. Layers of rock and sediment are deposited over millions of years to create a volumetric cube of rock with porosity that varies according to the geometries of the layers. Given this forward model, we wish to condition on a given volume, and recover a description of the individual layers that built up the volume.

Figure 7 shows the model, which generates a rock volume by first sampling parametric descriptions for each layer (the details are not shown; layers are described as a weighted set of basis functions), and then "rendering" them into a porosity volume (with Gaussian noise added at each voxel). The `render_layers` function encapsu-

A layer model of rock porosity

```
function dp_render_rock( data )
  num_layers = poissrnd();
  dp_sample_layer = dpmem( @sample_layer, 1.0 );
  for i=1:num_layers
    layer(i,:) = dp_sample_layer();
  end;
  rock_volume = render_layers( layers );
  data = rock_volume + randn( size(rock_volume) );
return;
```

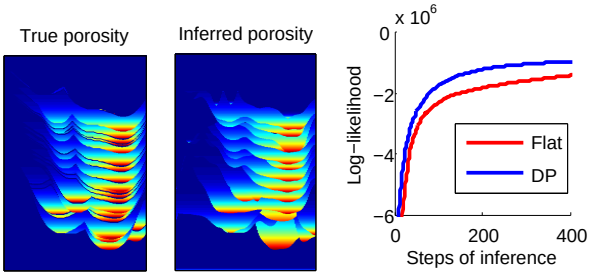

Figure 7: Results on modeling rock porosity.

lates the MEX/GPU-based renderer. Given this model, we perform inference using tempered MCMC, conditioning on the data shown in Fig. 7. A sample from the posterior shows a reconstructed volume (also shown in Fig. 7) that closely matches the true volume—the program has successfully inferred a parametric form for each layer.

The program calls a function named `dpmem`, a stochastic memoizer similar to that found in Church. `dpmem` accepts a function representing a base measure and a concentration parameter, and returns a new function which samples according to a Dirichlet process (DP). We use this to create a DP in "layer space," where parameters of layers can be shared. Fig. 7 (lower left) shows a comparison of inference with and without the `dpmem`: shared layers enable the inference engine to find higher-likelihood models faster. Hierarchical DPs are equally easy: we could (for example) let `f=dpmem(@randn,1.0)`, and `g=dpmem(@f,1.0)` to create an HDP with a Gaussian base measure. We consider such flexibility in defining models—combined with the rich libraries of Matlab—to be the major virtue of our approach.

4 RELATED WORK

Our MCMC algorithm is a new implementation of the Church MCMC algorithm [4], and is similar to the algorithm used in BLOG [7]. However, we provide a simpler and more uniform approach to constructing such algorithms for new languages, with small, clean code bases.

The closest related work is that of Kiselyov and Shan [6], who share the goal of transforming standard languages into probabilistic versions with little interpretive overhead. They use delimited continuations to control random choices without explicit choice naming. However, their technique is only applicable (prima facie) to enumeration or importance sampling, not MCMC.

Other related approaches include BUGS [16], HBC [1], and Infer.NET [8]. BUGS is not transformational (it simply loads a data structure into memory from a config file), but the implementations of both Infer.NET and HBC are comparable to our approach. Infer.NET uses a complicated transformation to compile csoft to C#. HBC compiles from a config file to C, but the specification language used by HBC is not a full-fledged language. We consider our approach simpler and more general.

With respect to modeling flexibility, our approach provides a simple dynamic knowledge-based model construction, which allows unbounded models, including complex recursions, to be handled efficiently. Models such as the naive implementation of the geometric distribution in Bher (Alg. 5), the `gamma/randn` example in Stochastic Matlab (Alg. 4), and nonparametric distributions such as the Dirichlet Process (Alg. 7) cannot be specified in these other languages.

A performance comparison between these languages is important, but beyond the scope of this paper. A careful comparison must account for issues such as mixing rates, initialization values, differences in parameters of proposal kernels, etc. and so we have explicitly limited this paper to an explanation of the technique, with comparisons reserved for a longer future paper.

5 CONCLUSION

We have described a lightweight implementation technique for probabilistic programming languages. The method is simple and fast, and the resulting languages permit a great deal of flexibility in the specification of distributions by mixing stochastic and deterministic elements with arbitrary language features (such as objects, inheritance, operator overloading, closures, recursion, libraries, etc.). Our example implementations have compact code bases and reasonable inference speed.

The main directions for improvement are better mixing and faster inference. Because source-to-source transformations are often compositional, more transformations could be applied to enhance performance. These could reduce redundant computation between traces, generate compound proposals, or implement constraint propagation for initializing conditioners. Future work will investigate these issues, as well as the possibility of new languages, such as stochastic Python.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of ONR, the James S. McDonnell Foundation and AFOSR grant FA9550-07-1-0075.

References

[1] H. Daume III. Hbc: Hierarchical bayes compiler, 2007.

[2] J. W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.

[3] A. Ghuloum. Ikarus scheme user's guide, 2008.

[4] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008.

[5] K. Kersting and L. D. Raedt. Bayesian logic programming: Theory and tool. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[6] O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384, 2009.

[7] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1352–1359, 2005.

[8] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[9] S. Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996.

[10] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based on sampling functions. *ACM Trans. Program. Lang. Syst.*, 31(1):1–46, 2008.

[11] A. Pfeffer. IBAL: A probabilistic rational programming language. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence*, pages 733–740. Morgan Kaufmann Publ., 2001.

[12] D. Poole. The independent choice logic and beyond. pages 222–243, 2008.

[13] A. Radul. Report on the probabilistic language scheme. Technical Report MIT-CSAIL-TR-2007-059, Massachusetts Institute of Technology, 2007.

[14] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[15] T. Sato and Y. Kameya. Prism: A symbolic-statistical modeling language. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1997.

[16] D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks. Bugs - bayesian inference using gibbs sampling version 0.50, 1995.