Composing inference algorithms as program transformations

Robert Zinkov
Indiana University
Bloomington, IN 47408 USA
zinkov@iu.edu

Chung-chieh Shan
Indiana University
Bloomington, IN 47408 USA
ccshan@indiana.edu

Abstract

Probabilistic inference procedures are usually coded painstakingly from scratch, for each target model and each inference algorithm. We reduce this effort by generating inference procedures from models automatically. We make this code generation modular by decomposing inference algorithms into reusable program-to-program transformations. These transformations perform exact inference as well as generate probabilistic programs that compute expectations, densities, and MCMC samples. The resulting inference procedures are about as accurate and fast as other probabilistic programming systems on real-world problems.

1 INTRODUCTION

Writing inference algorithms for probabilistic models is tedious and error-prone. Conceptually, these algorithms are combinations of simpler operations, such as computing the density of a distribution at a given point. So it is unfortunate that these algorithms are traditionally implemented from scratch. In this paper, we show how to describe these building blocks in code, so that they need not be rewritten for every new inference algorithm or model.

We contribute the first method for composing multiple inference algorithms over the same model, even exact and approximate ones over the same factor. Our approach is to express inference in terms of operations that transform one probabilistic program into another. We use probabilistic programs to represent distributions, though our approach is compatible with other representations such as factor graphs. The goal of our transformations is to turn a program that denotes a model into another program that, when interpreted to draw a weighted sample, is equivalent to the desired inference algorithm.

Because the output of an inference transformation is still a probabilistic program, we can apply further inference transformations to the program. In this way, we can subject a single model to multiple inference methods without coding them from scratch. We thus reduce the informal problem of combining inference methods to the formal and more automatable problem of composing program transformations. In particular, approximate inference methods can be composed with taking advantage of exact mathematical equivalences such as conjugacy.

2 MOTIVATION AND RELATED WORK

Developing inference algorithms that work on a variety of models has long been a goal of probabilistic inference, including graphical models and probabilistic programming. The composability of inference algorithms has unfortunately lagged behind the composability of models.

Many probabilistic programming systems allow a choice of inference methods, both exact and approximate. For example, the probabilistic language Church (Goodman et al. 2008) has many interpreters, each of which implements a different inference method. Systems such as Figaro, Factorie, Anglican and Wolfe (Pfeffer 2009; McCallum et al. 2008; Wood et al. 2014; Riedel et al. 2014) also allow adding inference methods, as new code in the host language where the systems are embedded. However, the end result of applying these inference methods is behavior or code in a different language, no longer a probabilistic program. Thus, it is difficult in these systems to apply one method to the result of another method.

More similar to our approach is the approach of Ścibior et al.'s (2015; 2016). Like us, they express and compose inference methods as transformations that produce probabilistic programs in the same language. Thus for example they reuse Sequential Monte Carlo to implement Particle Independent MH. But because their probabilistic language reuses many primitives from the host language Haskell, their transformations cannot inspect most of the input code, notably deterministic computations and code under a binding. In contrast, we can perform exact inference (Section 5), we can compute densities and conditional distributions (Section 4.3) in the face of deterministic dependencies, and we can generate MH samplers (Section 4.5) using a variety of proposal distributions.

So in general, transformations need to take programs as input as well as produce them as output in order to support the variety of inference composition found in the literature. To illustrate the need and the variety, below we recall some patterns of inference composition where we want to *reuse* existing implementations in ways unsupported by existing systems such as those named above.

Sometimes, we apply approximate inference to a model, then post-process the results using exact inference. For example, a popular way to perform inference for latent Dirichlet allocation (LDA) is to use Gibbs sampling (Griffiths and Steyvers 2004) to infer topic markers for each word, then infer from these topic markers the exact distribution on words given each topic.

Other times, we apply exact inference to parts of our model, and use an approximate method for the rest. As an example, Hughes et al. (2015) develop an inference algorithm for hierarchical Dirichlet processes that samples the truncation dictating the number of topics then performs variational inference for the other model parameters. This inference combination requires that sampling a truncation still leave in place a representation on which we can perform variational inference.

Another composition pattern emerges from recent work on parallelizing an inference algorithm to run on multiple machines (Neiswanger et al. 2014; Xu et al. 2014; Gelman et al. 2014). The pattern is to transform a posterior distribution for a parameter given the data into a model that lets us infer noisy versions of the parameter given subsets of the data. We then combine these noisy parameter estimates to infer the underlying parameter.

Finally, given a linear sequential model, we often want to predict future states of the system and the dynamics that govern them. Given the dynamics, for systems like Kalman filters, we may use exact inference to derive the state transition functions in closed form. Learning the dynamics, on the other hand, is usually treated as an approximate inference problem where we sample different possible dynamics given some observed states. This joint learning and exact inference again composes two inference algorithms. Section 6.1 shows our inference composition at work in a tiny instance of this case. We illustrate our approach using this example in Section 3.

3 EXAMPLE OF INFERENCE COMPOSITION

We illustrate our program-transformation approach to inference composition using a simple linear dynamical system. Our model below defines a joint distribution:

$$\begin{aligned}
&\mathrm{noise}_T && \sim \mathrm{Uniform}(3,8)\\
&\mathrm{noise}_E && \sim \mathrm{Uniform}(1,4)\\
&x_1 \mid \mathrm{noise}_T && \sim \mathrm{Normal}(0,\mathrm{noise}_T)\\
&m_1 \mid x_1, \mathrm{noise}_E && \sim \mathrm{Normal}(x_1,\mathrm{noise}_E)\\
&x_2 \mid x_1, \mathrm{noise}_T && \sim \mathrm{Normal}(x_1,\mathrm{noise}_T)\\
&m_2 \mid x_2, \mathrm{noise}_E && \sim \mathrm{Normal}(x_2,\mathrm{noise}_E)
\end{aligned}$$

We would like to draw samples from the posterior distribution over noise_T and noise_E given observations m_1 and m_2, using a Metropolis-Hastings (MH) sampler.

We start by representing the model in our language:

```
kalman =
 noiseT <~ Uniform(3, 8);
 noiseE <~ Uniform(1, 4);
 x1     <~ Normal( 0, noiseT);
 m1     <~ Normal(x1, noiseE);
 x2     <~ Normal(x1, noiseT);
 m2     <~ Normal(x2, noiseE);
 Dirac(((m1, m2), (noiseT, noiseE)))
```

The use of `Dirac` at the bottom shows that this distribution ranges over pairs of pairs of reals.

We first apply the *disintegration* transformation to get another program. As detailed in Section 4.3, disintegration takes as input a joint distribution and produces a program representing a family of posterior distributions. The new program is a function from the observations to the posterior distribution. For example, disintegrating `kalman` produces the program `kalman2` below. It takes as input `(m1, m2)` and returns the distribution over `(noiseT, noiseE)` given those values for `m1` and `m2`.

```
kalman2 = Lam((m1,m2),
 noiseT <~ Uniform(3, 8);
 noiseE <~ Uniform(1, 4);
 x1     <~ Normal( 0, noiseT);
 x2     <~ Normal(x1, noiseT);
 Weight( exp(-(m2-x2)^2/(2*noiseE^2))
           /noiseE/sqrt(2*pi)
         * exp(-(m1-x1)^2/(2*noiseE^2))
           /noiseE/sqrt(2*pi)
         , (noiseT, noiseE) ))
```

The use of `Lam` at the top and `Weight` at the bottom shows that this is a function from pairs of reals `(m1, m2)` to measures over pairs of reals `(noiseT, noiseE)`.

The next step is to apply the *simplification* transformation to `kalman2` to get `kalman3`.

```
kalman3 = Lam((m1,m2),
 noiseT <~ Uniform(3, 8);
 noiseE <~ Uniform(1, 4);
 Weight(P, (noiseT, noiseE)))
```

This program is equivalent to `kalman2`, except the simplification transformation has symbolically integrated out the `Normal`-distributed random variables `x1` and `x2` and replaced them by an observation likelihood in closed form, which we elide above as P.

We next apply to `kalman3` another program transformation we call *mh*, which implements MH sampling. The mh transformation takes as input two programs. The first program represents a proposal distribution, or more precisely a function from the current sample to a distribution over proposed samples. Here we use a proposal distribution that with equal probability resamples one of `noiseT` and `noiseE` while keeping the other fixed:

```
proposal = Lam((noiseT, noiseE),
 Superpose((1/2, n <~ Uniform(3, 8);
                 Dirac((n, noiseE))),
           (1/2, n <~ Uniform(1, 4);
                 Dirac((noiseT, n)))))
```

The second input to the mh transformation represents the target distribution. In this example, it is the part of `kalman3` above after the top line `Lam((m1,m2),`. From these inputs, the mh transformation computes a symbolic formula for the MH acceptance ratio and embeds it in a program representing a transition kernel. The new program, `kalman4` below, is a function from the current sample to a distribution over pairs of proposed samples and acceptance ratios:

```
kalman4 = Lam((noiseT, noiseE),
 proposed <~ Superpose(
   (1/2, n <~ Uniform(3, 8);
         Dirac((n, noiseE))),
   (1/2, n <~ Uniform(1, 4);
         Dirac((noiseT, n)))),
 Dirac((proposed, A)))
```

The elided part A is a symbolic formula that computes the acceptance ratio using the current sample `(noiseT, noiseE)` and the sample `proposed` by the `Superpose`. This acceptance ratio can then be used to decide whether to accept or reject the `proposed`.

We then perform further optimizations on `kalman4`, including algebraic simplifications and rewriting the program to use fewer `<~`s. We describe in more detail the kinds of optimizations we perform in Section 5. The resulting program, `kalman5`, has the following structure:

```
kalman5 = Lam((noiseT, noiseE),
 Superpose(
  (1/2, n <~ Uniform(3, 8);
        Dirac(((n, noiseE), A_T))),
  (1/2, n <~ Uniform(1, 4);
        Dirac(((noiseT, n), A_E)))))
```

The elided parts A_T and A_E are algebraically simplified formulas for the acceptance ratio in each of the two cases.

Finally we feed this last program `kalman5` to a sampler (Algorithm 1). Given an observation and a current sample, this sampler produces a proposed sample and the MH acceptance ratio of that sample.

```
sample(App(App(kalman5,(0,1)),(5,2)),
       [])
>>> (((5,1.6811397),0.7924639),1.0)
```

In the command above, `(0,1)` is the observation, and `(5,2)` is the current sample. In the output above, `(5,1.6811397)` is the proposed sample, and `0.7924639` is its acceptance ratio.

4 INFERENCE METHODS AS PROGRAM TRANSFORMATIONS

To compose inference methods, we pose them as transformations of one probabilistic program into another. We then achieve the desired inference method for the former program by applying a simpler inference method, such as exact inference or weighted sampling, to the latter program. For example, in Section 3 we feed a program to disintegration (Section 4.3), then mh (Section 4.5), then simplification (Section 5), and finally sampling. Only in the final sampling step is any random choice made!

We first define our probabilistic language, then describe various program transformations that work in concert.

4.1 LANGUAGE DESCRIPTION

Below is our core grammar of probabilistic programs:

$$
\begin{aligned}
e \ ::=\ & x \mid 1 \mid e - e \mid e < e \mid \mathtt{exp}(e) \mid \mathtt{If}(e,e,e) \mid \ldots \\
\mid\ & \mathtt{Sum}(e,e,x,e) \mid \mathtt{Int}(e,e,x,e) \\
\mid\ & \mathtt{Lam}(x,e) \mid \mathtt{App}(e,e) \mid (e,e) \mid e\mathtt{[0]} \mid e\mathtt{[1]} \\
\mid\ & \mathtt{Uniform}(e,e) \mid \mathtt{Normal}(e,e) \\
\mid\ & \mathtt{Gamma}(e,e) \mid \mathtt{Weight}(e,e) \\
\mid\ & \mathtt{Categorical}((e,e),\ \ldots) \\
\mid\ & \mathtt{Superpose}((e,e),\ \ldots) \mid x \mathtt{<\sim} e \mathtt{;}\ e
\end{aligned}
$$

The first line of this grammar says that our language includes ordinary programming support for variables, math, and `If`. The second line adds primitives to represent `Sum`mation and `Int`egration, used in Section 4.2. The third line adds functions and tuples.

The remainder of the grammar is what makes our language probabilistic: we add primitives that represent and compose measures. To start with, `Uniform(1,2)` represents the uniform distribution over real numbers between 1 and 2, and `Normal(3,4)` represents the normal distribution with mean 3 and standard deviation 4.

`Weight(1,8)` represents the probability distribution that assigns its entire probability mass 1 to the single outcome 8. We write `Dirac(8)` as syntactic sugar for it. In contrast, `Weight(0.7,8)` represents the measure, or unnormalized distribution, that assigns the probability 0.7 to the single outcome 8. This primitive lets our language represent (unnormalized) measures in general, not just (normalized) probability distributions. This expressivity lets us separately reuse a transformation that produces an unnormalized measure (Section 4.3) and a transformation that subsequently normalizes a measure (Section 4.4). Also, `Weight` lets us represent a distribution by combining a base measure and a density function.

`Categorical` represents the categorical distribution with a sequence of zero or more pairs. The first element of each pair is the probability of selecting the outcome that is the second element of the pair. If the first elements of the pairs do not sum to 1, they are normalized.

`Superpose` is like `Categorical`, except it does not normalize, so it can represent measures in general. We can define `Superpose` in terms of `Categorical` and `Weight`, but it is actually more convenient to define `Weight` and `Categorical` in terms of `Superpose`.

The final primitive `<~` (pronounced "bind") composes two distributions `e1` and `e2`. The second distribution `e2` may depend on the outcome `x` of `e1`. The outcome of the composed distribution `x<~e1;e2` is the outcome of `e2`. This primitive lets our language represent sequential and hierarchical models. A simple example is this model:

$$x \sim \text{Uniform}(0,2) \qquad y|x \sim \text{Uniform}(x,3)$$

We write the marginal distribution over y as

```
x <~ Uniform(0, 2); Uniform(x, 3)
```

and the joint distribution over (x, y) as

```
x <~ Uniform(0, 2);
y <~ Uniform(x, 3); Dirac((x,y))
```

To make the semantics of our language more concrete, Algorithm 1 shows a sampler that takes a probabilistic program as input and returns a draw from the distribution it represents. It is our only operation that calls a random number generator. We apply it last in a sequence of transformations to perform approximate inference.

Like a typical interpreter, Algorithm 1 takes as input not only a program but also an environment, which is a table mapping variable names to values. Also, because our language includes unnormalized measures, this sampler returns not only a draw but also an importance weight.

Algorithm 1: Weighted sampler: sample(m, *env* = [])

Input: program representing a measure: m
Input: environment: *env*
Output: pair of values (outcome, weight)
Examine m
if *m has the form* `Weight(`w_1`,`e_1`)` **then**
  Evaluate e_1 in the environment *env*, obtaining v_1
  Return (v_1, w_1)
else if *m has the form* `Normal(`e_1`,`e_2`)` **then**
  Evaluate e_1 in the environment *env*, obtaining v_1
  Evaluate e_2 in the environment *env*, obtaining v_2
  Sample from the normal distribution with mean v_1 and standard deviation v_2, obtaining v_3
  Return $(v_3, 1)$
else if *m has the form* x`<~`m_1`;`m_2 **then**
  Call Algorithm 1 on m_1 with the environment *env*, obtaining (v_1, w_1)
  Let *env*$'$ be the environment *env* extended with x having the value v_1
  Call Algorithm 1 on m_2 with the environment *env*$'$, obtaining (v_2, w_2)
  Return $(v_2, w_1 \cdot w_2)$
else
  The other cases are similar to above
end

4.2 EXPECTATION TRANSFORMATION

The rest of this section describes various inference transformations that we apply to our probabilistic programs. Because we implement some transformations in terms of others, we describe the transformations not in the order we apply them but in the order we implement them.

Our expectation transformation turns any program that represents a distribution into another program that represents its expected value. This transformation is exact and simple even though the expected values of many distributions have no closed form, because our language represents integrals symbolically with `Int`. For example, the expectation transformation turns the program `x<~Uniform(0,2); Uniform(x, 3)` into

```
Int(0,2,x, Int(x,3,y, y)/(3-x))/(2-0)
```

The latter program represents the integral $\frac{1}{2-0}\int_0^2 \frac{1}{3-x}\int_x^3 y\,dy\,dx$. To compute this integral in closed form is to perform exact inference on the given distribution. The expectation transformation itself does not do so; nor does it approximate the integral by sampling.

Algorithm 2: Expectation transformation: expect(m, f)

Input: program representing a measure: m
Input: program representing a function: f
Output: program representing a number
Examine m
if *m has the form* `Weight` (w_1, e_1) **then**
  Return $w_1 \cdot$ `App` (f, e_1)
else if *m has the form* `Normal` (e_1, e_2) **then**
  Return `Int` ($-\infty$, ∞, `x`, $e_3 \cdot$ `App` (f, `x`))
  where the program e_3 computes the density of the `Normal`(e_1, e_2) distribution at `x`
else if *m has the form* x`<~`m_1`;`m_2 **then**
  Call Algorithm 2 with m_2 and f obtaining e_3
  Call Algorithm 2 with m_1 and `Lam` (x, e_3)
else
  The other cases are similar to above
end

Specified more generally, the expectation transformation turns any program that represents a measure, along with a function from the sample space to numbers, into another program that represents the integral of the given function with respect to the given measure. We show this transformation as Algorithm 2. It handles primitive distributions such as `Normal` by looking up their density from a table.

4.3 DENSITY AND DISINTEGRATION

Turning a distribution into its density function is naturally expressed as a program transformation (Bhat et al. 2012, 2013). More precisely, the density transformation takes as input a probabilistic program representing a distribution, and returns another program representing a function that maps each point in the sample space to the density at that point. Note that this transformation does not compute any density numerically. It only returns a program that computes densities when interpreted by our weighted sampler (Algorithm 1). For example, the density transformation turns the probabilistic program

```
x <~ Uniform(0, 2);
y <~ Uniform(x, 3); Dirac((x, y))
```

into the density function `Lam((x,y), If(0<x<2, If(x<y<3, 1/(3-x), 0)/(2-0), 0))`.

We implement density in terms of another program transformation, *disintegration* (Shan and Ramsey 2017; Narayanan and Shan 2017). Disintegration is similar to conditioning in that it takes a probabilistic program representing a joint distribution $\Pr(X, Y)$ as input, but instead of returning a conditional distribution $\Pr(Y \mid X = x)$, disintegration returns an unnormalized slice $\Pr(Y, X = x)$ of the original distribution. More precisely, disintegration returns a program representing a function from values of x to measures $\Pr(Y, X = x)$. Such a (measurable) function is also known as a *kernel*.

Algorithm 3: Density transformation: density(m, t)

Input: program representing a measure: m
Input: program representing value drawn from m: t
Output: program representing a nonnegative number

1. Disintegrate `x<~`m`; Dirac((x,Unit))`, obtaining e_1
2. Call Algorithm 2 on `App(`e_1`,`t`)` and `Lam(y,1)`

Algorithm 4: Observation transformation: observe(m, t)

Input: program representing a measure: m
Input: program representing value drawn from m: t
Output: program representing a measure
Examine m
if *m has the form* `Uniform` (e_1, e_2) *or* `Normal` (e_1, e_2) *or* `Gamma` (e_1, e_2) **then**
  Let d be a program that computes the density of the distribution m
  Return `Weight(App(`d`,`t`), ` t`)`
else if *m has the form* x`<~`m_1`;`m_2 **then**
  Call Algorithm 4 recursively with m_2 and t obtaining m_3
  Return x`<~`m_1`;`m_3
else
  Raise an error about not being able to handle m
end

Taking advantage of the fact that disintegration does not normalize the measures it returns, we implement the density transformation in terms of disintegration and expectation. This implementation is shown in Algorithm 3. It invokes disintegration (letting Y be the space that consists of a single point `Unit`) then expectation (letting the integrand f be the function that maps `Unit` to 1).

Disintegration is useful independently of the density transformation. For example, Section 3 uses it to turn the prior `kalman` into the posterior `kalman2`.

We sketch how disintegration works in terms of a simpler program transformation, which we call *observation* (Algorithm 4). This transformation takes as input a measure m and a value t that could have been drawn from m, and returns a measure which only returns t, weighted by how likely that value was to be drawn from m. For example, the observation transformation turns the program

```
x <~ Uniform(0, 2); Uniform(x, 3)
```

and the variable `y` into the program

```
x <~ Uniform(0, 2);
Weight(If(x<y<3, 1/(3-x), 0), y)
```

Algorithm 5: Normalization transformation: normalize(m)

Input: program representing a measure: m
Output: program representing a probability distribution

1. Call Algorithm 2 on m and `Lam(x,1)` obtaining the program e_1
2. Return `x<~`m`; Weight(1/`e_1`, x)`

As indicated at the bottom in Algorithm 4, the observation transformation only handles a subset of our language. In particular, it does not handle `Dirac`, so it does not handle the typical program `kalman` in Section 3. In general, if the input program performs arithmetic or any other deterministic computation to produce the observation t, then we need to invert this deterministic computation and insert any Jacobian factors required. This inversion is what the disintegration provides over observation.

To relate observation and disintegration more precisely, suppose the program m represents a measure over X, the program e represents a value in Y, and observation turns m and `x` into m_1. Then disintegrating the program `...; x<~`m`; Dirac((x,`e`))` yields a program equivalent to `Lam(x, ...; dummy<~`m_1`; Dirac(`e`))`.

4.4 NORMALIZATION AND CONDITIONING

The presence of `Weight` in our language enables the observation and disintegration transformations to return measures that are typically unnormalized. To recover a probability distribution, we must reweight the measure. We define this *normalization* operation as a program transformation as well, shown as Algorithm 5.

Conditioning can now be defined by composition: it is just disintegration, followed by normalizing the measure.

4.5 MCMC SAMPLING TRANSFORMATIONS

A major contribution of this paper is to implement Markov chain Monte Carlo (MCMC) methods, such as MH sampling and Gibbs sampling, in a way that applies to a variety of target distributions and composes with other inference techniques. We express an MCMC method as a transformation from a program representing the target distribution to a program representing the transition kernel. Whereas the transformation itself makes no random choices, the latter program can be interpreted by our weighted sampler (Algorithm 1) to generate a random chain, or subject to simplification (Section 5).

Following this approach, our MCMC implementations closely resemble their textbook presentation. As shown in Algorithm 6, where the textbook presentation of the acceptance ratio refers to the target and proposal densities, our implementation invokes the density transformation (Algorithm 3) on two probabilistic programs, representing the target and proposal distributions. Using the fact that the density transformation symbolically handles free variables such as `old` and `new`, we perform the transformation just once (not once per sampler iteration) to generate a program that takes the current state as input.

Algorithm 6: Metropolis-Hastings sampling transformation: mh(*proposal*, *target*)

Input: program representing the proposal kernel: *proposal*
Input: program representing the target distribution: *target*
Output: program representing MCMC transition kernel with acceptance ratio

1. Let `old` and `new` be fresh variable names
2. Call Algorithm 3 on *target* and `old`, obtaining p_{old}
3. Call Algorithm 3 on *target* and `new`, obtaining p_{new}
4. Call Algorithm 3 on `App(`*proposal*`,new)` and `old`, obtaining $q_{\text{old;new}}$
5. Call Algorithm 3 on `App(`*proposal*`,old)` and `new`, obtaining $q_{\text{new;old}}$
6. Let e_1 be $(p_{\text{new}} \cdot q_{\text{old;new}})/(p_{\text{old}} \cdot q_{\text{new;old}})$
7. Return `Lam(old, new<~App(`*proposal*`,old); Dirac((new, `e_1`)))`

Algorithm 7: Gibbs sampling transformation: gibbs(*target*)

Input: program representing the n-dimensional target distribution: *target*
Output: program representing MCMC transition kernel
Let $\mathbf{x}$ be the set of the n variables in the *target*
Initialize *choices* to the empty sequence `[]`
For each $x_i \in \mathbf{x}$:

1. Let x_{-i} be the rest of the variables
2. Let e_1 be $\mathbf{x}$`<~`*target*`; Dirac((`x_{-i}`,`x_i`))`
3. Disintegrate e_1, obtaining e_2
4. Let e_3 be `App(`e_2`, `x_{-i}`)`
5. Call Algorithm 5 on e_3, obtaining e_4
6. Let $\mathbf{y}$ be $\mathbf{x}$ except replacing x_i by `new`
7. Let e_5 be `new<~`e_4`; Dirac(`$\mathbf{y}$`)`
8. Add the pair `(1/`n`, `e_5`)` to *choices*

Return `Lam(`$\mathbf{x}$`, Superpose(`*choices*`))`

Gibbs sampling is a special case of MH, where the proposal kernel combines the results of conditioning the target distribution along each dimension. The acceptance ratio is then always 1, so it need not be computed. To produce such a proposal kernel automatically, we implement Gibbs sampling as a separate transformation, Algorithm 7. The input is a program representing an n-dimensional distribution $\Pr(x_1, \ldots, x_n)$. For each random variable x_i, we condition (Section 4.4) on the other variables x_{-i} to get a program that resamples x_i. We then combine these n programs to form the proposal kernel.

5 SIMPLIFICATION

Because we express each inference technique as a transformation that produces a probabilistic program, rather than as an interpreter that makes immediate random choices, we can optimize and simplify the produced programs. To this end, we apply the optimizations discussed by Carette and Shan (2016). This *simplification* transformation does not change the measure represented by a program but tries to place the program in a form that, when interpreted by our weighted sampler (Algorithm 1), draws samples faster and with more uniform weights.

Based on computer algebra, the simplification transformation recognizes conjugacy relationships, integrates out latent variables, and performs algebraic simplifications. Like other transformations, simplification operates on a program before any variables receive their values, so in particular its efficacy is independent of data sizes. The rest of this section briefly describes these optimizations.

Conjugacy relationships Simplification recognizes when a density represented by `Weight` matches the density of a primitive distribution. A simple example arises from the joint distribution $\Pr(Y, X)$ represented below:

```
x <~ Normal(a, s);
y <~ Normal(x, t); Dirac((y, x))
```

Disintegrating this program (Section 4.3) produces

```
x <~ Normal(a, s);
Weight( exp(-(y-x)^2/(2*t^2))
        /t/sqrt(2*pi)          , x )
```

This latter program scales the measure `Normal(a,s)` with the density `exp(...)/t/sqrt(2*pi)` to represent the conditional distribution $\Pr(X \mid Y)$ up to a constant factor. Normalizing and simplifying it yields

```
Normal( (y*s^2+a*t^2)/(s^2+t^2),
        s*t/sqrt(s^2+t^2) )
```

using the conjugacy relationship between `Normals` (assuming `s` and `t` are positive). This simplified program runs faster; it draws samples without weighting them.

This optimization is symbolic, in the sense that it works even when the initial program contains free variables such as `a`, `s`, and `t`, whose values are unknown.

This optimization is robust because it recognizes not words like `Normal` but the densities they denote. Thus it works even if we express `Normal(0,1)` by spelling out its density, whether we expand the polynomial `-(y-x)^2`. All conjugacies among `Normal`, `Gamma`, and `Beta` thus fall out from recognizing their densities.

Integrating out a variable When a distribution is described using a latent random variable, it usually helps to eliminate the variable. Such latent variables include `x1` and `x2` in Section 3, as well as `x` in

```
x <~ Normal(0, 1); Normal(x, 1)
```

The simplification transformation eliminates these variables. In particular, it integrates out continuous latent variables symbolically. The density-recognition machinery just described then produces simpler, faster, and equivalent programs, such as `Normal(0,sqrt(2))`.

This integration is symbolic, again in the sense that it works even when the initial program contains free variables whose values are unknown. For example, the program `x <~ Normal(a, s); Normal(x, t)` simplifies to `Normal(a, sqrt(s^2+t^2))`.

Algebraic simplifications When we produce a program that calculates acceptance ratios, the numerator and denominator share many factors, which are usually canceled out by hand. The simplification transformation automates this optimization using computer algebra, so an expression like `(`a`*`b`)/(`a`*`c`)` becomes b/c.

6 EXPERIMENTAL RESULTS

To demonstrate that our approach is modular and practical, we apply multiple inference methods (MH, Gibbs) to a variety of models. We conduct three experiments using the Hakaru system (Narayanan et al. 2016).

Modular means we can re-use the components in Section 4 and Section 5 to produce all three samplers. In each experiment, a pipeline composed of reusable inference transformations turns a concise generative model into an executable MCMC sampler in seconds.

Practical means our approach can solve real-world problems by expressing popular models and inference methods discussed in the literature. The largest of our three experiments is the third, a document classification task using the 20 Newsgroups corpus.

We measure the accuracy and speed of our automatically generated samplers, showing they are in line with solutions from commonly used probabilistic programming languages. Our samplers are more accurate across the board because simplification eliminates all latent continuous variables, regardless of the dimensionality of the problem (that is, input and output array sizes).

Table 1: MH sampler run times for linear dynamics

Inference method	**Run time** (msecs) Mean	SD
WebPPL	1078	16
Hakaru without simplifications	1321	93
Hakaru with simplifications	269	10
Handwritten	207	4

Table 2: MH sampler ESS rates for linear dynamics

Inference method	**ESS per sample** noise_T	noise_E
WebPPL	0.03	0.01
Hakaru	0.09	0.34

All measurements were produced on a quad-core Intel i5-2540M processor running 64-bit Ubuntu 16.04. Our samplers use Glasgow Haskell Compiler 8.0.1 `-O2`.

6.1 MH SAMPLING FOR DYNAMICS

In our first experiment, we use MH to sample the random parameters of the linear dynamical system in Section 3. We compare our generated samplers with one produced by WebPPL, a state-of-the-art probabilistic programming system, and with one written by hand. The WebPPL sampler was compiled to JavaScript using Node 0.10.28.

Table 1 shows that our system generates a fast sampler, measured by using each sampler to draw 20,000 samples 10 times. Thanks to the simplifications that turn `kalman4` into `kalman5`, the Hakaru sampler is 4 times as fast as the WebPPL sampler for the conditional distribution `kalman2`. (These times exclude the few seconds each system takes to compile the model into a sampler.)

Table 2 shows that our samplers generate good samples, quantified by the Effective Sample Size (ESS). Our ESS is higher per sample compared to WebPPL, because latent variables have been integrated out in `kalman3`.

6.2 GIBBS SAMPLING FOR CLASSIFICATION

In our second and third experiments, we generate Gibbs samplers and compare them to JAGS (v4.20), a probabilistic programming system widely considered practical for Gibbs sampling. We measure accuracy by how well the samplers recover true classifications, and speed by the time it takes to produce samples. This time consists of *initialization* time and time spent actually sampling. Initialization time is the time a system takes from receiving the model to generating the first sample: for JAGS to load the model into memory, and for Hakaru to simplify the model and compile the result into machine code.

Our second experiment is to classify synthetic data using a Gaussian mixture model that has 3 components.

Figure 1 shows that Hakaru requires fewer sweeps than JAGS to achieve the same accuracy. Each curve plots the accuracy of one chain over the course of 15 sweeps on 250 data points. After just one sweep, all 20 Hakaru chains are $> 50\%$ accurate, unlike the 20 JAGS chains, which take a few sweeps to catch up. The cause is that Hakaru's simplification transformation recovers a collapsed Gibbs sampler that computes the sample mean and variance of each mixture component in closed form.

Figure 2 shows Hakaru is about one order of magnitude slower than JAGS, measured by how long 6 sweeps take, varying data size from 500 to 2500 points. The top two curves represent two samplers generated by Hakaru with different lower-level optimizations: the second-from-top curve adds a *histogram* optimization to compute summary statistics such as the per-mixture-component sum $\sum_{i=1}^{N} \left\{ \begin{smallmatrix} t_i & \text{if } z_i = z^* \\ 0 & \text{otherwise} \end{smallmatrix} \right.$ in a single pass over the data for all components z^*. The bottom two curves show the run time of JAGS, with and without initialization. JAGS's speed advantage can be explained by Hakaru's current inability to reuse computation between updates during a sweep. Still, Hakaru is practical for this real-world task.

In our third experiment, Hakaru generates a classifier for the 20 Newsgroups corpus that is more accurate than JAGS and comparable in speed. We use the same Multinomial Naive Bayes model and 20 Newsgroups corpus as McCallum and Nigam (1998). We hold out 10% of the labels and use Gibbs sampling to infer them. We evaluate the samplers on data sizes ranging from 200 to all 19997 documents, evenly distributed among newsgroups.

Because the sampler generated by Hakaru is collapsed, it is more accurate than JAGS in two ways. First, Figure 3 shows Hakaru achieves better accuracy than JAGS after one sweep, and continues to for at least 1000 sweeps. Each curve there plots the accuracy (moving average with window size 20) of one chain on 400 documents. Second, Figure 4 shows Hakaru more accurate than JAGS across data sizes. We use 2 sweeps there since JAGS does not perform above chance with only 1 sweep.

Figure 5 shows Hakaru is as fast as JAGS, measured by how long 2 sweeps take, varying data size. JAGS's initialization time grows with the data size, while Hakaru's is constant. Whereas JAGS unrolls loops into a pointer-based stochastic graph whose size grows with the data, Hakaru generates tight loops over unboxed arrays irrespective of the data size. Even disregarding initialization time, JAGS is at best 4 times faster than Hakaru.

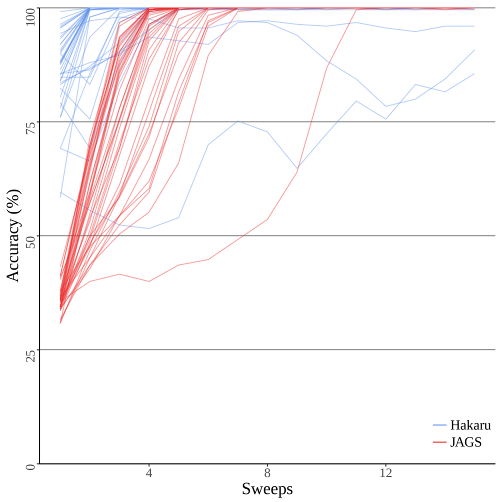

Figure 1: Gibbs sampler accuracy for Gaussian mixture

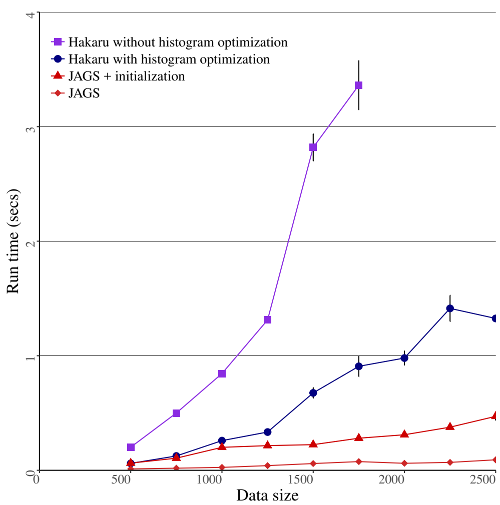

Figure 2: Gibbs sampler run times for Gaussian mixture

7 CONCLUSIONS

We express inference methods by composing program transformations such as disintegration and expectation. The resulting modular inference procedures perform comparably to other probabilistic programming systems and are usable for practical problems. This technique makes it easier and faster to create and test inference procedures and to explore novel inference methods.

Acknowledgements

We thank David Belanger, Jacques Carette, and Chad Scherrer for helpful discussions, feedback, and suggestions. This research was supported by DARPA contract FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc., and the Indiana METACyt Initiative.

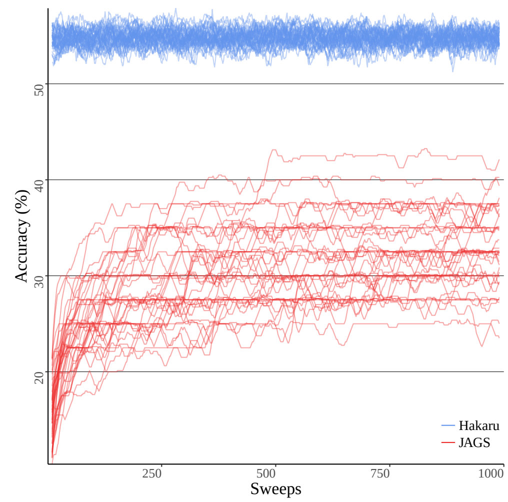

Figure 3: Document classification accuracy by sweeps

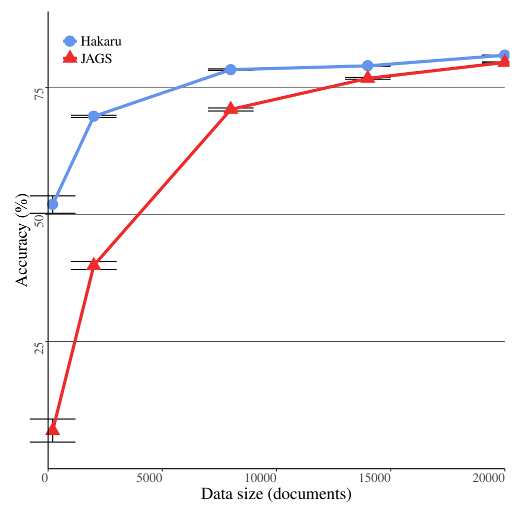

Figure 4: Document classification accuracy by data size

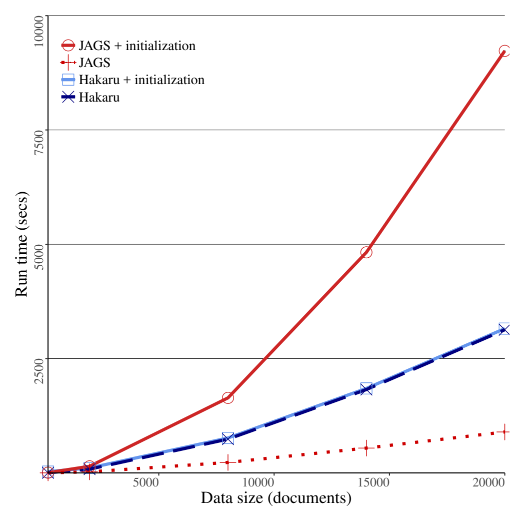

Figure 5: Document classification run times. Error bars were too small to display.

References

Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. A type theory for probability density functions. In *POPL '12: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 545–556. ACM Press, January 2012.

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

Jacques Carette and Chung-chieh Shan. Simplifying probabilistic programs using computer algebra. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, pages 135–152, 2016.

A. Gelman, A. Vehtari, P. Jylänki, C. Robert, N. Chopin, and J. P. Cunningham. Expectation Propagation as a Way of Life. *ArXiv e-prints*, December 2014.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5228–5235, 2004.

Michael Hughes, Dae Il Kim, and Erik Sudderth. Reliable and scalable variational inference for the hierarchical Dirichlet process. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 370–378, 2015.

Andrew McCallum and Kamal Nigam. A comparison of event models for naive Bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

Andrew McCallum, Khashayar Rohanemanesh, Michael Wick, Karl Schultz, and Sameer Singh. Factorie: Efficient probabilistic programming for relational factor graphs via imperative declarations of structure, inference and learning. In *NIPS Workshop on Probabilistic Programming*, 2008.

Praveen Narayanan and Chung-chieh Shan. Symbolic conditioning of arrays in probabilistic programs. In *ICFP '17: Proceedings of the ACM International Conference on Functional Programming*. ACM Press, 2017.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In Oleg Kiselyov and Andy King, editors, *Proceedings of FLOPS 2016: 13th International Symposium on Functional and Logic Programming*, number 9613 in Lecture Notes in Computer Science, pages 62–79. Springer, 2016.

Willie Neiswanger, Chong Wang, and Eric Xing. Asymptotically exact, embarrassingly parallel MCMC. In *The Conference on Uncertainty in Artificial Intelligence (UAI)*, 2014.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

Sebastian Riedel, Sameer Singh, Vivek Srikumar, Tim Rocktäschel, Larysa Visengeriyeva, and Jan Noessner. WOLFE: Strength Reduction and Approximate Programming for Probabilistic Programming. In *International Workshop on Statistical Relational AI (StarAI)*, 2014.

Adam Ścibior and Zoubin Ghahramani. Modular construction of Bayesian inference algorithms. In *NIPS Workshop on Advances in Approximate Bayesian Inference*, 2016.

Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 165–176, 2015.

Chung-chieh Shan and Norman Ramsey. Exact Bayesian inference by symbolic disintegration. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 130–144. ACM Press, 2017.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

Minjie Xu, Balaji Lakshminarayanan, Yee Whye Teh, Jun Zhu, and Bo Zhang. Distributed Bayesian posterior sampling via moment sharing. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3356–3364, 2014.