From high-level inference algorithms to efficient code

RAJAN WALIA and PRAVEEN NARAYANAN, Indiana University, USA
JACQUES CARETTE, McMaster University, Canada
SAM TOBIN-HOCHSTADT and CHUNG-CHIEH SHAN, Indiana University, USA

Probabilistic programming languages are valuable because they allow domain experts to express probabilistic models and inference algorithms without worrying about irrelevant details. However, for decades there remained an important and popular class of probabilistic inference algorithms whose efficient implementation required manual low-level coding that is tedious and error-prone. They are algorithms whose idiomatic expression requires random array variables that are *latent* or whose likelihood is *conjugate*. Although that is how practitioners communicate and compose these algorithms on paper, executing such expressions requires *eliminating* the latent variables and *recognizing* the conjugacy by symbolic mathematics. Moreover, matching the performance of handwritten code requires speeding up loops by more than a constant factor.

We show how probabilistic programs that directly and concisely express these desired inference algorithms can be compiled while maintaining efficiency. We introduce new transformations that turn high-level probabilistic programs with arrays into pure loop code. We then make great use of domain-specific invariants and norms to optimize the code, and to specialize and JIT-compile the code per execution. The resulting performance is competitive with manual implementations.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; • **Computing methodologies → Symbolic calculus algorithms**; • **Mathematics of computing → Integral calculus**; **Probabilistic representations**; **Variable elimination**; **Gibbs sampling**; *Metropolis-Hastings algorithm.*

Additional Key Words and Phrases: probabilistic programs, arrays, plates, multidimensional distributions, marginalization, conjugacy, map-reduce, loop optimization, collapsed Gibbs sampling

ACM Reference Format:
Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From high-level inference algorithms to efficient code. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2019), 30 pages.

1 SIMPLIFYING AND OPTIMIZING PROBABILISTIC PROGRAMMING

Many users of an algorithm would rather not worry about the details of its efficient implementation or correctness proof. Whether the algorithm is copied from a textbook by a programmer or generated from a domain-specific language by a compiler, the vocabulary used to express the algorithm needs to be mapped to executable code before the algorithm can be run. For example, if the algorithm invokes sorting, then it is easier to turn into executable code using a language or library that features a sorting routine. To take a more recent example, if the algorithm refers to the gradient of a function, then it is easier to turn into executable code using automatic differentiation.

Authors' addresses: Rajan Walia, rawalia@indiana.edu; Praveen Narayanan, pravnar@umail.iu.edu, Department of Computer Science, Indiana University, Bloomington, USA; Jacques Carette, carette@mcmaster.ca, Department of Computing and Software, McMaster University, Canada; Sam Tobin-Hochstadt, samth@cs.indiana.edu; Chung-chieh Shan, ccshan@indiana.edu, Department of Computer Science, Indiana University, Bloomington, USA.

2475-1421/2019/1-ART1
https://doi.org/

In the realm of probabilistic programming, while a wide variety of languages [Carpenter et al. 2017; De Raedt et al. 2007; de Salvo Braz et al. 2007; Fischer and Schumann 2003; Goodman et al. 2008; Goodman and Stuhlmüller 2014; Huang et al. 2017; Kiselyov 2016; Kiselyov and Shan 2009; Lunn et al. 2000; Mansinghka et al. 2014; Milch et al. 2007; Narayanan et al. 2016; Nori et al. 2014; Patil et al. 2010; Pfeffer 2007, 2016; Tran et al. 2017; Tristan et al. 2014; Wood et al. 2014; Wu et al. 2016] have made many algorithms easier to express, many practically-important inference methods continue to require manual transformation and implementation. In this paper, we extend the range of probabilistic inference algorithms that can be turned automatically into executable code, to include *arrays* whose distributions need to be *simplified* and whose loops need to be *optimized*.

- Simplification includes *eliminating latent variables* and *recognizing conjugate likelihoods*.
  - Briefly, a latent variable is a random variable whose value may be used in the program but is not returned. Elimination is widely applied to discrete and continuous variables [de Salvo Braz et al. 2007; Dechter 1998; Poole and Zhang 2003; Sanner and Abbasnejad 2012; Zhang and Poole 1994, 1996] and is known in various contexts as *Rao-Blackwellization* [Blackwell 1947; Casella and Robert 1996; Gelfand and Smith 1990; Kolmogorov 1950; Murray et al. 2018; Rao 1945], *collapse* [Koller and Friedman 2009; Liu 1994; Liu et al. 1994; Venugopal and Gogate 2013], *marginalization* [Meng and van Dyk 1999; Obermeyer et al. 2018], and *integrating out* [Griffiths and Steyvers 2004; Resnik and Hardisty 2010].
  - Briefly, a conjugate likelihood is a weight on samples that can be made constant while preserving semantics by changing how the samples are generated in the first place. Conjugacy is a preferred starting point and basic building block of Bayesian data modeling [Gelman et al. 2014, page 36] and underlies such popular applications as Naive Bayes classification [Bayes 1763] and Bayesian linear regression [Borgström et al. 2016].
- Loop optimization includes reordering sums to achieve superlinear speedups, and fusing and specializing loops to obtain one more order of magnitude in performance.

As the description above suggests, the importance of this class of algorithms has been established in applied statistics for decades. However, turning the vocabulary used to express them into executable code had required manual calculation and coding that is tedious and error-prone [Cook et al. 2006; Geweke 2004]. Our work thus paves the way for programmers and compilers alike to target a higher-level probabilistic language with arrays and to worry less about the details of the correctness of distribution simplifications and the efficiency of loop optimizations.

One major reason that turning high-level algorithms into efficient code is difficult—whether by hand or by machine—is that it requires sophisticated symbolic mathematics. Recent research has started to automate such reasoning on probabilistic programs [Carette and Shan 2016; Gehr et al. 2016; Hoffman et al. 2018; Tran et al. 2017]. However, even systems that support arrays either fail to perform popular transformations such as latent-variable elimination (as in Augur [Tristan et al. 2014], AugurV2 [Huang et al. 2017], and Edward [Tran et al. 2017]) or unroll random choices entirely at prohibitive performance cost (as in PSI [Gehr et al. 2016]). Given that arrays are key in almost any inference algorithm, unrolling is a non-starter for efficient execution.

We remove these obstacles to automation. Probabilistic programmers can express high-level algorithms and expect sophisticated transformations to automate efficient execution on large arrays of data. We present a domain-specific compilation pipeline that meets all these goals. Specifically:

(1) We extend probabilistic programs and their simplification to those with arrays of large or arbitrary size, such as arrays of size n or of size n_1-by-n_2-by-n_3, where each n is large and/or unknown (Section 3). Our array simplification transformation is modular in that it reuses existing technology underlying scalar simplification and, like that technology, eschews brittle pattern matching of specific distributions and extends easily to new primitive distributions.

- We extend symbolic integration in computer algebra to high- and arbitrary-dimensional integrals, such as integrals over $\mathbb{R}^n$ or over $\mathbb{R}^{n_3 n_2 n_1}$, where each n is large and/or unknown.
- We introduce the symbolic *unproduct* operation to uncover independence underlying a program so as to apply our simplification transformation. This process traverses an input term systematically and recursively to uncover its equivalence to a sequence of products $\prod_i \prod_j \prod_k$ of any given length.

(2) We introduce the *histogram* optimization (Section 4), which asymptotically speeds up loops by rewriting them as map-reduce expressions in a modular and general way.
- This optimization *unnests* loops, by locating conditionals buried deep inside any level of nested loop bodies. It is particularly effective on simplified array probabilistic programs.

(3) We optimize the resulting array-manipulating code aggressively yet safely, by taking advantage of the domain-specific features of probabilistic programs (Section 5).
- We carefully engineer loop-invariant code motion (LICM) and loop fusion, so that they apply soundly, widely, and profitably.
- We further use just-in-time (JIT) compilation to propagate static information.

(4) We show that while each of our techniques is valuable, their composition—our *pipeline*—is dramatically more effective. In other words, each bullet item above is a significant and essential contribution.

Section 2 lays out our compilation pipeline and sets the stage for these technical contributions.

We emphasize that our aim is not to improve the compilation of models already handled by existing systems, but rather to enable the compilation of algorithms not handled by existing systems and not expressed by previous probabilistic programmers. We compile probabilistic programs that directly and concisely express a new and open class of algorithms of lasting and current significance that previously required manual, tedious, and error-prone mathematics and coding. Of course, we can only measure our system against other systems on tasks that they can also do. The quantitative evaluation in Section 6 demonstrates that our proof-of-concept system achieves the competitive performance expected of the newly expressed algorithms, relative to handwritten code for the same algorithms and other state-of-the-art systems carrying out different algorithms. Whereas our system automates exact inference and collapsed Metropolis-Hastings (MH) sampling, our modular tools and techniques automate tasks often performed manually by practitioners of many other inference methods. Hence, for example, it is promising to incorporate our contributions in a future system that supports Hamiltonian Monte Carlo (HMC) or variational inference.

2 COMPILATION PIPELINE OVERVIEW

The heart of many popular inference algorithms is to calculate a conditional distribution exactly and possibly sample from it. This pattern is clearest and most challenging in Gibbs sampling, which repeatedly updates a sample by conditioning on some of its dimensions. But the same pattern recurs in MH, HMC, and importance sampling, because they require computing a density, which is the total of a conditional distribution. And in important cases such as Bayesian linear regression, an exact solution is available, because conditioning the model on the observed data results in a distribution that can be represented in a closed form. Due to this pattern, practitioners communicate and compose not only probabilistic models (such as hidden Markov models [Rabiner 1989]) but also inference algorithms (such as Markov Chain Monte Carlo (MCMC) [MacKay 1998]) using the same probabilistic programming constructs: sampling, sequencing, looping, conditioning, and so on. Therefore, it makes sense to express both modeling and inference distributions in a single declarative language of measures, as pioneered by the proof-of-concept probabilistic programming system Hakaru [Narayanan et al. 2016; Zinkov and Shan 2017].

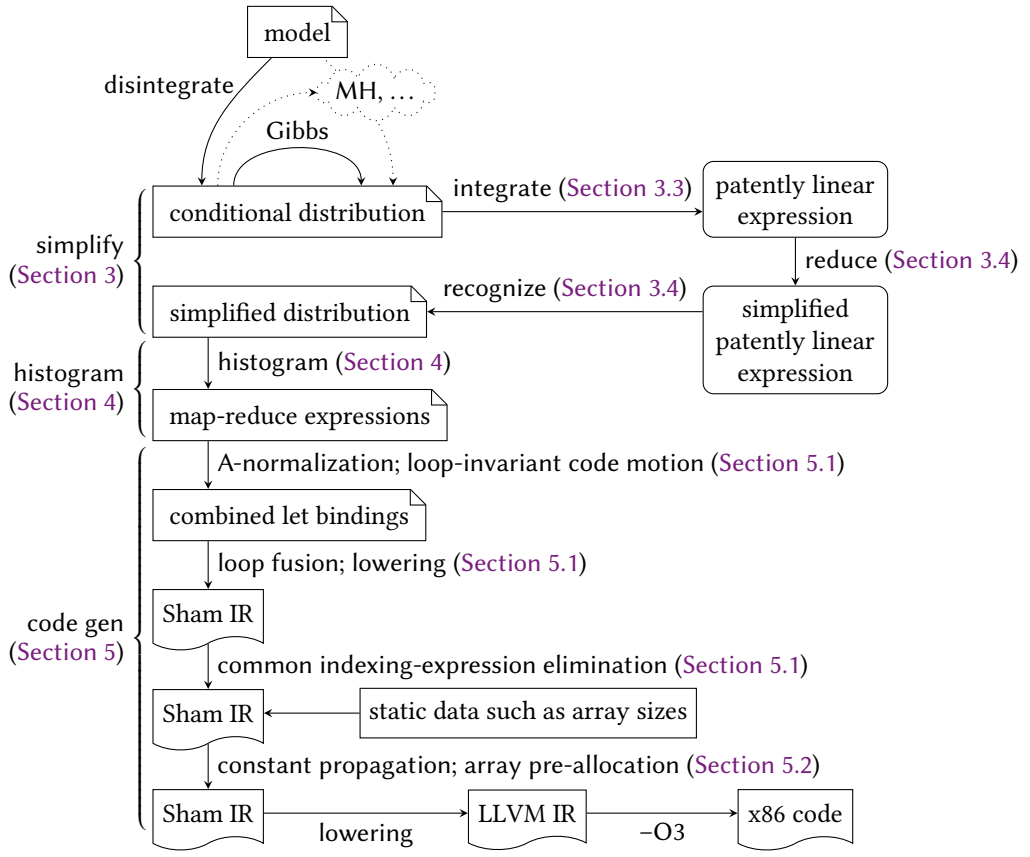

Fig. 1. Our pipeline, compiling probabilistic programs via math into imperative code to process data

The compilation pipeline in this paper is designed to express such inference algorithms concisely and execute them efficiently. The starting point is a probabilistic program that expresses the desired inference algorithm by denoting the conditional distribution to calculate and possibly sample. That is, we represent the inference distribution as a *generative* process, which is a step-by-step procedure for drawing random variables and computing a final outcome. Some procedures score their outcome so its *importance weight* varies from run to run; other procedures make no random choice so the computation is deterministic.

Figure 1 lays out our compilation pipeline from model to code. Because this paper starts with the conditional distribution near the top, it leaves open the issue of how to find the desired inference algorithm. After all, there is no single inference method that works well for all models, and knowing what works well takes domain expertise not available to a compiler. In Hakaru, which our implementation builds on, the inference distribution is typically produced by metaprogramming constructs that form a directed graph of choices [Narayanan and Shan 2017; Shan and Ramsey 2017; Zinkov and Shan 2017], depicted schematically at the very top of the figure. In another context, the inference distribution may be produced by hand. Either way, producing the inference distribution is outside the scope of this paper. Rather, the contributions of this paper start as soon as the inference algorithm is expressed as a conditional distribution, as it is naturally in the literature.

The probabilistic IR in which we express and simplify inference distributions is the Hakaru language. Because Hakaru eschews general recursion and is typed and terminating, all abstractions can be beta-reduced away near the start of the pipeline, leaving a first-order core whose constructs express mathematical operations and the measure monad (Figure 2). Other probabilistic languages that allow general recursion may well profit from selectively applying our pipeline, but that is outside the scope of this paper.

An elementary tour. We sketch a simple application to give an impression of the parts of our pipeline. The rest of the paper elaborates on this *Gaussian mixture* example.

Suppose we observe n data points. Each data point lies at some location along the real line and belongs to one of m classes, so we store our observations in two arrays of size n: the locations in $\vec{s} \in \mathbb{R}^n$ and the class labels in $\vec{y} \in \{0, \ldots, m-1\}^n$. A simple model of how these data points came to be might say that each class has an underlying location, not directly observed, and the location of each data point is a noisy measurement of the underlying location of the class of the data point. Whether we are interested in estimating the locations of the classes or predicting the locations of upcoming data points, the *disintegration* transformation can produce a probabilistic program that denotes the distribution of our quantity of interest, conditioned on our observations. In this program, each possible underlying location of a class is weighted, or scored, by the likelihood of the noisy measurements that we observed from the class.

Starting with this probabilistic program for a conditional distribution, the *simplification* transformation produces a closed-form formula that (a) estimates the underlying location of each class or predicts the location of each upcoming data point, and (b) evaluates how well the observed data fits the model. As one might expect, part (a) is based on the mean of the data points in that class, and part (b) is based on the variance of the data points in that class. Thus, the formula generated by simplification contains sums such as

$$\sum_{j=0}^{n-1} \begin{cases} \vec{s}[j]^2 & i{=}\vec{y}[j] \\ 0 & \text{otherwise} \end{cases} \qquad \sum_{j=0}^{n-1} \begin{cases} \vec{s}[j] & i{=}\vec{y}[j] \\ 0 & \text{otherwise} \end{cases} \qquad \sum_{j=0}^{n-1} \begin{cases} 1 & i{=}\vec{y}[j] \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $i \in \{0, \ldots, m-1\}$ is a class label. Each of the three sums take time $O(n)$ to compute, so looping over all m classes takes $O(mn)$ time. Improving upon this situation, the *histogram* transformation discovers that each of the three sums can be computed for all m classes in a single pass through the data that creates an array of size m:

```
let hist₂ := newArray(m)        let hist₁ := newArray(m)        let hist₀ := newArray(m)
for j = 0 to n − 1:             for j = 0 to n − 1:             for j = 0 to n − 1:
  hist₂[y⃗[j]] += s⃗[j]²           hist₁[y⃗[j]] += s⃗[j]             hist₀[y⃗[j]] += 1
```
(2)

Now the computation takes only $O(n)$ time for all m classes. Even better, the code in (2) never materializes because our domain-specific code generator aggressively optimizes these loops: it fuses the three loops into one, eliminates the common indexing expressions $\vec{y}[j]$ and $\vec{s}[j]$, pre-allocates the three output arrays, and JIT-specializes the machine code not only to the given size m but also to the *addresses* of the pre-allocated output arrays.

These aggressive optimizations might be overkill if we only need to perform the computation once for a given size m. But the generated code may well be the inner loop of an approximate inference algorithm that solves a harder problem. For example, suppose that the class labels $\vec{y}$ are not observed. Then, a closed-form solution is no longer available. A popular solution approach, called MCMC, is to design a random walk among the m^n possible values of $\vec{y}$ that approximates the target distribution. The transition probabilities of this random walk are calculated by repeating part (b) above at every step. Hence these optimizations are worthwhile. We have automated them.

3 SIMPLIFYING ARRAY PROGRAMS

To *simplify* a probabilistic program is to produce a more efficient (or readable) program while still representing the same distribution. Carette and Shan [2016] introduced a simplifier that applies computer algebra strategically to the linear operator denoted by a probabilistic program: their simplifier eliminates latent variables and recognizes conjugate likelihoods by exploiting domain constraints. We extend that simplifier to handle probabilistic programs with arrays, which naturally represent high- and arbitrary-dimensional distributions that arise in inference algorithms.

Our extended simplifier handles latent variables and conjugacy by exploiting constraints on array indices. A key part, the *unproduct* operation (Section 3.4), uncovers independence in the mathematical denotations of array programs; this operation is derived from first principles and subsumes AugurV2's rewrite rule for indirect indexing [Huang et al. 2017]. Without unrolling an array or even knowing its concrete size, our simplifier computes exact distribution parameters that recover sufficient statistics such as sample mean, sample variance, and word counts by document class. These informative symbolic parameters let us compile inference algorithms such as MCMC on Dirichlet-multinomial mixtures.

Simplification depends heavily on computer algebra. Our extended simplifier is implemented in Maple, but we do not rely on features specific to Maple, and we have experimented with SymPy and obtained promising results.

The rest of this section uses a progression of examples to explain what our extended simplifier does, why it's useful, and how it works. To pump intuition about Bayesian inference, these examples use simplification as a form of exact inference, even though simplification is also essential for efficient approximate inference, as discussed in Section 2.

3.1 Background

We tour Carette and Shan's simplifier [2016] with an example. Consider the distribution over $\mathbb{R}^2$ generated by

(1) drawing $x \in \mathbb{R}$ from the normal distribution with some fixed mean μ and standard deviation 1;
(2) drawing $y, z \in \mathbb{R}$ from the normal distribution with mean x and standard deviation 1; and
(3) returning the pair $[y, z]$.

These steps model two noisy measurements y, z of the unknown location x of a particle along the real line. To model that we do not directly observe the location x, the returned outcome $[y, z]$ omits x, and we say that the random variable x is *latent*. We represent this distribution by the term

$$\mathsf{Bind}(\mathsf{Gaussian}(\mu, 1), x, \mathsf{Bind}(\mathsf{Gaussian}(x, 1), y, \mathsf{Bind}(\mathsf{Gaussian}(x, 1), z, \mathsf{Ret}([y, z])))), \tag{3}$$

in which μ is a free variable, and x, y, z are bound and take scope to their right. To create generative processes, we use two constructs of Giry's monad of probability distributions [1982] (which was popularized by Ramsey and Pfeffer [2002]):

- $\mathsf{Ret}(e)$ produces the outcome e deterministically.
- $\mathsf{Bind}(m, x, m')$ carries out the process m (such as the primitive distribution $\mathsf{Gaussian}(\mu, 1)$) and binds the outcome to the variable x then carries out m' to get the final outcome.

Figure 2 shows the essential part of the language. We write the informal type $\mathbb{M}\,T$ for distributions (measures) over the type T. Whereas Giry's and Ramsey and Pfeffer's works concerned probability distributions, our language includes the $\mathsf{Weight}(e, m)$ construct for weighting samples or scaling distributions. Because the weight e is not bounded, our language can express not just probability or sub-probability distributions but the more general class of *s-finite* measures [Staton 2017].

One way to interpret the term (3) is as a monadic program that samples three random numbers each time it is run. But before running the program, we can first use Carette and Shan's simplifier

Types $$T, U ::= \mathbb{R} \mid \mathbb{R}^+ \mid \mathbb{Z} \mid \mathbb{N} \mid \mathbb{M}\,T \mid \mathbb{A}\,T \mid \cdots$$

Some primitive distributions (see [Carette and Shan 2016] for more)

$$\frac{a : \mathbb{R} \quad b : \mathbb{R}}{\mathsf{Uniform}(a, b) : \mathbb{M}\,\mathbb{R}} \qquad \frac{\mu : \mathbb{R} \quad \sigma : \mathbb{R}^+}{\mathsf{Gaussian}(\mu, \sigma) : \mathbb{M}\,\mathbb{R}} \qquad \frac{\alpha : \mathbb{R}^+ \quad \beta : \mathbb{R}^+}{\mathsf{Beta}(\alpha, \beta) : \mathbb{M}\,\mathbb{R}^+} \qquad \frac{e : \mathbb{A}\,\mathbb{R}^+}{\mathsf{Categorical}(e) : \mathbb{M}\,\mathbb{N}}$$

Measure combinators

$$\frac{e : T}{\mathsf{Ret}(e) : \mathbb{M}\,T} \qquad \frac{e : \mathbb{R}^+ \quad m : \mathbb{M}\,T}{\mathsf{Weight}(e, m) : \mathbb{M}\,T} \qquad \frac{m : \mathbb{M}\,T \quad \begin{array}{c}[x : T]\\ \vdots\\ m' : \mathbb{M}\,U\end{array}}{\mathsf{Bind}(m, x, m') : \mathbb{M}\,U}$$

Array constructs

$$\frac{e_0 : T \quad \ldots \quad e_{n-1} : T}{[e_0, \ldots, e_{n-1}] : \mathbb{A}\,T} \qquad \frac{e : \mathbb{A}\,T \quad i : \mathbb{N}}{e[i] : T} \qquad \frac{n : \mathbb{N} \quad \begin{array}{c}[i : \mathbb{N}]\\ \vdots\\ e : T\end{array}}{\mathsf{ary}(n, i, e) : \mathbb{A}\,T} \qquad \frac{n : \mathbb{N} \quad \begin{array}{c}[i : \mathbb{N}]\\ \vdots\\ m : \mathbb{M}\,T\end{array}}{\mathsf{Plate}(n, i, m) : \mathbb{M}\,(\mathbb{A}\,T)} \qquad \frac{e : \mathbb{A}\,T}{\#e : \mathbb{N}}$$

Fig. 2. Informal term typing rules for distributions and (new) for arrays. The bracketed judgments indicate the scope of bound variables; for example, in $\mathsf{Bind}(m, x, m')$, the variable x takes scope over m' but not m.

[2016] to turn it into

$$\mathsf{Bind}(\mathsf{Gaussian}(\mu, \sqrt{2}), y, \mathsf{Bind}(\mathsf{Gaussian}(\tfrac{1}{2}(\mu + y), \tfrac{\sqrt{6}}{2}), z, \mathsf{Ret}([y, z]))). \tag{4}$$

The latent variable x was eliminated, and the distributions of y and z adjusted accordingly. Compared to the program (3), the program (4) makes fewer random choices yet produces the same distribution. That is, the two programs are equivalent if we interpret $\mathbb{M}$ as the distribution monad, but (4) uses randomness more efficiently if we interpret $\mathbb{M}$ as the sampling monad [Ramsey and Pfeffer 2002]. Moreover, we can read off from the form of (4) exactly how to perform a kind of Bayesian *inference*: If we have measured y but not z, we can predict z using

$$\mathsf{Gaussian}(\tfrac{1}{2}(\mu + y), \tfrac{\sqrt{6}}{2}), \tag{5}$$

a subterm of (4). (In particular, we can estimate z using the mean $\frac{1}{2}(\mu + y)$.) That is, the simplifier has computed (5) to be the *conditional* distribution of z given y in our model.

To pump intuition about Bayesian inference, we ordered the random variables x, y, z in (3) so that simplification produces a conditional distribution (5). If we had commuted the bindings of y and z, then simplification would instead produce the conditional distribution of y given z. This illustrates that simplification, like a typical optimization pass, is sensitive to syntactic choices in semantically equivalent inputs, even though it preserves semantics.

We now zoom into how simplification works. Figure 1 illustrates the structure of Carette and Shan's simplifier [2016], whose parts we extend with arrays. It turns (3) into (4) by three steps.

First, the simplifier converts the program (3) into

$$\int_{\mathbb{R}} \int_{\mathbb{R}} \int_{\mathbb{R}} \frac{\mathrm{e}^{-\frac{1}{2}(x-\mu)^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(y-x)^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(z-x)^2}}{\sqrt{2\pi}} h([y, z])\, dz\, dy\, dx. \tag{6}$$

This quantity is the *expectation* of an arbitrary measurable function $h : \mathbb{R}^2 \to \mathbb{R}^+$ with respect to the distribution. In other words, the simplifier interprets $\mathbb{M}$ as the expectation monad [Ramsey and Pfeffer 2002]. The expectation (6) is linear in h. To understand this integral, consider when $h([y, z]) = \begin{cases} 1 & [y,z] \in S \\ 0 & \text{otherwise} \end{cases}$ for some $S \subseteq \mathbb{R}^2$; the integral is then just the probability of S. Each factor in (6), such as $\frac{\mathrm{e}^{-\frac{1}{2}(x-\mu)^2}}{\sqrt{2\pi}}$, is the density of a primitive distribution, here $\mathsf{Gaussian}(\mu, 1)$ at x.

$$g ::= \hbar(e) \mid e \cdot g \mid g_1 + \cdots + g_n \mid \mathsf{If}(e, g, g) \mid \int_a^b g\, dx \mid \int_X g\, d\vec{x} \qquad X ::= (a, b) \mid \prod_{i=c}^{d} X$$

Fig. 3. The grammar of expressions patently linear in $\hbar$. The denotation of g and the range of $\hbar$ lie in $\mathbb{R}^+$. Metavariables a, b, c, d, e stand for expressions, whereas $\hbar, x, i$ stand for variables. New is the last g-production, for integrals over high- and arbitrary-dimensional spaces X. We omit $g ::= \sum_{i=a}^{b} g$ as we treat distributions over $\mathbb{Z}$ by analogy to those over $\mathbb{R}$.

Second, noticing that the variable x is latent (that is, no argument to h contains x free), the simplifier symbolically integrates over x to get

$$\int_{\mathbb{R}} \int_{\mathbb{R}} \frac{\mathrm{e}^{-\frac{1}{3}\mu^2} \mathrm{e}^{-\frac{1}{3}y^2} \mathrm{e}^{\frac{1}{3}\mu y} \mathrm{e}^{-\frac{1}{3}z^2} \mathrm{e}^{\frac{1}{3}\mu z} \mathrm{e}^{\frac{1}{3}yz}}{2\sqrt{3}\pi} h([y, z])\, dz\, dy. \tag{7}$$

Third, inverting the first step, the simplifier converts (7) back to a program, namely (4). This conversion requires the simplifier to recognize that a factor, such as the fraction in (7), is the density of a primitive distribution, here (5) at z. Recognizing a factor as the density of a distribution subsumes recognizing the conjugacy of a likelihood with respect to a distribution. To recognize a factor by matching it against syntactic patterns would be brittle and ad-hoc. Instead, the simplifier characterizes the factor $f(z)$ by its *holonomic* representation [Chyzak and Salvy 1998; Wilf and Zeilberger 1992], a first-order linear differential equation (here $3f'(z) = (\mu + y - 2z)f(z)$) whose coefficients (here 3 and $\mu + y - 2z$) are polynomials in z.

Fortunately, functions with holonomic representations constitute a large class with useful closure properties, such as closure under integration, differentiation, and composition with algebraic functions [Kauers 2013]. Taking advantage of these closure properties, the simplifier computes the holonomic representation from the factor expression compositionally and not by pattern matching. Moreover, because the coefficients are polynomials, their ratios can be matched efficiently and robustly using existing algorithms such as Euclid's algorithm. Therefore, this third and final step of the simplifier is robust against syntactic perturbations, general across primitive distributions, and modular so that implementing each primitive distribution separately suffices for conjugacy relationships among them to be recognized [Carette and Shan 2016].

The first of the three steps, integrate(m, h), produces an expression patently linear in h by structural induction on the program m. The expression produced by integrate(m, h) denotes the *expectation*, or *Lebesgue integral*, of the function h with respect to the distribution m; for example, integrate((3), h) produces (6), and integrate((4), h) produces something that expands to (7). Because distributions m and the linear operators $\lambda h.\, \text{integrate}(m, h)$ are in one-to-one correspondence [Pollard 2001, Section 2.3], any simplification of integrate(m, h) that preserves its meaning also preserves the distribution denoted by m. But feeding (6) willy-nilly to a computer algebra system will not out-of-the-box improve it to (7) and may even make it worse. Instead, Carette and Shan's simplifier [2016] operates strategically on parts of a patently linear expression, guided by the grammar in Figure 3.

3.2 Scalar simplification is not enough

Given that Carette and Shan's simplifier [2016] works on scalar probabilistic programs, one might hope that array probabilistic programs can be simplified by applying the same simplifier to scalars in loop bodies. Unfortunately, the array programs that express desired inference algorithms require extending the simplifier at the level of mathematical denotations, not just applying it strategically at the level of source programs. Before describing our extended simplifier, we motivate it with four increasingly tricky examples. Along the way, we introduce the array constructs of our language.

We begin with an example of an array program that is trivial to handle using the scalar simplifier. The distribution over $\mathbb{R}^2$ in Section 3.1 generalizes to one over $\mathbb{R}^{2n}$, generated by repeating the following for $i = 0, \ldots, n-1$:

(1) drawing $x \in \mathbb{R}$ from the normal distribution with some fixed mean μ and standard deviation 1;
(2) drawing $y, z \in \mathbb{R}$ from the normal distribution with mean x and standard deviation 1; and
(3) returning the pair $[y, z]$.

This distribution models $2n$ noisy measurements of the unknown locations of n particles along the real line. Because the loop body returns a pair of reals, the loop returns an array of n pairs of reals. We represent this distribution by

$$\mathsf{Plate}(n, i, \mathsf{Bind}(\mathsf{Gaussian}(\mu, 1), x, \mathsf{Bind}(\mathsf{Gaussian}(x, 1), y, \mathsf{Bind}(\mathsf{Gaussian}(x, 1), z, \mathsf{Ret}([y, z]))))). \tag{8}$$

The new construct Plate forms a monadic loop: the variable n above is free like μ and denotes an arbitrary iteration count, and the variable i is an index that takes scope over the monadic action to its right. In general, $\mathsf{Plate}(n, i, m)$ is a monadic action whose outcome is an array of n elements, independently drawn from the distributions $m\{i \mapsto 0\}, \ldots, m\{i \mapsto n-1\}$. (Indices begin at 0.) The informal typing rule for Plate in Figure 2 says accordingly that if m has type $\mathbb{M}\,T$ then $\mathsf{Plate}(n, i, m)$ has type $\mathbb{M}\,(\mathbb{A}\,T)$, where $\mathbb{A}\,T$ means arrays of T. This Plate construct is named after *plate notation* for repetition in Bayes nets [Buntine 1994; Koller and Friedman 2009]. It is like `Data.Vector.generateM` in Haskell, but since each array element is drawn independently, a Plate is a *parallel* comprehension [Huang et al. 2017].

Of course, we can apply the scalar simplifier to the subexpression (3) in (8), and the result is an improvement for the same reasons as for (3): it makes fewer random choices ($2n$ instead of $3n$) and enables probabilistic inference (from measuring each y to predicting each z).

But pointwise simplification is not enough. It is just as natural to express essentially the same distribution by multiple loops: we can generate a pair of arrays of n reals by

(1) drawing $\vec{x}[i] \in \mathbb{R}$ from the normal distribution with mean μ and standard deviation 1, for $i = 0, \ldots, n-1$;
(2) drawing $\vec{y}[i], \vec{z}[i] \in \mathbb{R}$ from the normal distribution with mean $\vec{x}[i]$ and standard deviation 1, for $i = 0, \ldots, n-1$; and
(3) returning the pair $[\vec{y}, \vec{z}]$.

We use accents on the three variables $\vec{x}, \vec{y}, \vec{z}$ to remind ourselves that they denote arrays, so their type is $\mathbb{A}\,\mathbb{R}$, and element i of $\vec{x}$ is $\vec{x}[i]$, not $x[i]$. Again using Plate, we represent this distribution by

$$\begin{aligned}&\mathsf{Bind}(\mathsf{Plate}(n, i, \mathsf{Gaussian}(\mu, 1)), \vec{x},\\ &\mathsf{Bind}(\mathsf{Plate}(n, i, \mathsf{Gaussian}(\vec{x}[i], 1)), \vec{y},\\ &\mathsf{Bind}(\mathsf{Plate}(n, i, \mathsf{Gaussian}(\vec{x}[i], 1)), \vec{z}, \mathsf{Ret}([\vec{y}, \vec{z}]))))\end{aligned} \tag{9}$$

and we want to simplify this probabilistic program to

$$\begin{aligned}&\mathsf{Bind}(\mathsf{Plate}(n, i, \mathsf{Gaussian}(\mu, \sqrt{2})), \vec{y},\\ &\mathsf{Bind}(\mathsf{Plate}(n, i, \mathsf{Gaussian}(\tfrac{1}{2}(\mu + \vec{y}[i]), \tfrac{\sqrt{6}}{2})), \vec{z}, \mathsf{Ret}([\vec{y}, \vec{z}]))).\end{aligned} \tag{10}$$

Before we can apply the scalar simplifier, we seem to have to first fuse the three Plate loops in (9), to form a single loop body to simplify.

Loop fusion is still not enough. Fusing loops may seem promising, but the following richer classic example illustrates the broader variety of array programs that simplification ought to improve. Suppose we would like to model $n + 1$ data points drawn from a *mixture* of m normal distributions.

Each component i of the mixture might represent a different subpopulation, such as researchers of different specialties. A *Gaussian mixture* distribution [Pearson 1894] can be generated by

(1) drawing the *mixture weights* $\vec{\theta}$, an array of m non-negative reals that sum to 1, from some *Dirichlet* distribution;
(2) drawing m *component means* $\vec{x}[i]$ from Gaussian(μ, σ), for $i = 0, \ldots, m-1$;
(3) drawing n class labels $\vec{y}[j] \in \{0, \ldots, m-1\}$ from the discrete distribution $\vec{\theta}$, for $j = 0, \ldots, n-1$;
(4) drawing n data points $\vec{s}[j]$ from Gaussian$(\vec{x}[\vec{y}[j]], 1)$, for $j = 0, \ldots, n-1$;
(5) drawing one more class label $z \in \{0, \ldots, m-1\}$ from the discrete distribution $\vec{\theta}$;
(6) drawing one more data point t from Gaussian$(\vec{x}[z], 1)$; and
(7) returning the tuple $[\vec{y}, \vec{s}, z, t]$.

By first drawing the random indices $\vec{y}, z$ then using those class labels to decide which means in $\vec{x}$ to draw $\vec{s}, t$ around, this process models how different subpopulations share different characteristics. Again, we want to automate the process by which human experts simplify this program to make fewer choices (eliminating $\vec{\theta}, \vec{x}$) and enable inference (predicting z, t from $\vec{y}, \vec{s}$).

We first examine how to eliminate $\vec{x}$, then turn to eliminating $\vec{\theta}$. At first glance, it is not obvious how to eliminate the latent array variable $\vec{x}$, because the loop where $\vec{x}$ is drawn and the loop where $\vec{x}$ is used (to draw $\vec{s}$) range over different domains ($i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$) and cannot be fused. However, we can group the iterations of the latter loop by which element of $\vec{x}$ they use: each $\vec{x}[i]$ is used to draw exactly those $\vec{s}[j]$ for which $i = \vec{y}[j]$. In other words, we can group the elements of $\vec{s}$ by how they are classified in $\vec{y}$. Hence we can transform steps 2, 4, and 6 above into a single loop that, informally speaking, repeats the following for $i = 0, \ldots, m-1$:

(2′) drawing $\vec{x}[i]$ from Gaussian(μ, σ);
(4′) drawing $\vec{s}[j]$ from Gaussian$(\vec{x}[i], 1)$, for each $j = 0, \ldots, n-1$ such that $i = \vec{y}[j]$; and
(6′) drawing t from Gaussian$(\vec{x}[i], 1)$ if $i = z$.

Because each $\vec{x}[i]$ drawn in the new step 2′ is used only in steps 4′ and 6′ in the same iteration over i and not beyond, scalar simplification can eliminate $\vec{x}[i]$. More formally, eliminating each $\vec{x}[i]$ requires performing the integral $\int_{\mathbb{R}} e^{f(\vec{x}[i])}\, d\vec{x}[i]$, whose integrand $e^{f(\vec{x}[i])}$ multiplies together the densities of the same Gaussian$(\vec{x}[i], 1)$ at all the elements of $\vec{s}$ whose classification in $\vec{y}$ is i. Because just one Gaussian$(\vec{x}[i], 1)$ is involved, the exponent

$$f(x) = \sum_{j=0}^{n-1} \begin{cases} -\frac{1}{2}(\vec{s}[j]-x)^2 & i=\vec{y}[j] \\ 0 & \text{otherwise} \end{cases} = -\frac{1}{2}\left(\sum_{j=0}^{n-1} \begin{cases} \vec{s}[j]^2 & i=\vec{y}[j] \\ 0 & \text{otherwise} \end{cases}\right) + x\left(\sum_{j=0}^{n-1} \begin{cases} \vec{s}[j] & i=\vec{y}[j] \\ 0 & \text{otherwise} \end{cases}\right) - \frac{1}{2}x^2\left(\sum_{j=0}^{n-1} \begin{cases} 1 & i=\vec{y}[j] \\ 0 & \text{otherwise} \end{cases}\right) \tag{11}$$

depends on just one element x of $\vec{x}$ at a time. The result of the integration is expressed in terms of the three summations in the right-hand side of (11) (same as (1)). They are the square-sum, sum, and count of just those elements of $\vec{s}$ labeled by $\vec{y}$ to belong to class i; these summations recover the sufficient statistics of the input data. Thus, simplifying array programs requires extracting per-element formulas such as (11) and conjuring the conditionals therein to preserve semantics.

Even iteration reordering is not enough. Grouping loop iterations in the source program is enough to eliminate the latent variable $\vec{x}$ but not $\vec{\theta}$. To explain why, we first need to explain what Dirichlet distributions are. Dirichlet distributions are a family of distributions over arrays of non-negative numbers that sum to 1 (the informal type is $\mathbb{M}\,(\mathbb{A}\,\mathbb{R}^+)$). For simplification, we expand step 1 above, "draw $\vec{\theta}$ from some Dirichlet distribution", as a macro to the following:

(1a) drawing $\vec{p}[i] \in [0, 1]$ from some *Beta* distribution, for $i = 0, \ldots, m-2$; and

(1b) returning the array $\vec{\theta} = [1 - \vec{p}[0],$
$\vec{p}[0] \cdot (1 - \vec{p}[1]),$
$\vec{p}[0] \cdot \vec{p}[1] \cdot (1 - \vec{p}[2]),$
$\vec{p}[0] \cdot \vec{p}[1] \cdot \vec{p}[2] \cdot (1 - \vec{p}[3]),$
$\ldots,$
$\vec{p}[0] \cdots \vec{p}[m-3] \cdot (1 - \vec{p}[m-2]),$
$\vec{p}[0] \cdots \vec{p}[m-3] \cdot \vec{p}[m-2]].$
(Here we notate an array by a bracketed list of elements.)

This expansion is a well-known, finite-dimensional variant of the *stick-breaking process* [Gelman et al. 2014, page 583]. The intuition behind the name is to start with a stick of length 1 and break off a piece of proportion $\vec{p}[0]$, then from that piece break off a piece of proportion $\vec{p}[1]$ (that is, of length $\vec{p}[0] \cdot \vec{p}[1]$), then from *that* piece break off a piece of proportion $\vec{p}[2]$ (that is, of length $\vec{p}[0] \cdot \vec{p}[1] \cdot \vec{p}[2]$), and so on.[1] We represent this process by the term

$$\mathsf{Bind}\big(\mathsf{Plate}\big(m-1, i, \mathsf{Beta}(\alpha(i)+1, \beta(i)+1)\big), \vec{p}, \mathsf{Ret}\big(\mathsf{ary}\big(m, i, \big(\textstyle\prod_{k=0}^{i-1} \vec{p}[k]\big) \cdot \left\{\begin{smallmatrix} 1-\vec{p}[i] & i<m-1 \\ 1 & i=m-1 \end{smallmatrix}\right.\big)\big)\big), \quad (12)$$

where $\alpha(i)+1$ and $\beta(i)+1$ are parameters of the Beta distribution that may depend on i. Here ary is an array comprehension construct: the term $\mathsf{ary}(m, i, e)$ denotes an array of size m whose elements are $e\{i \mapsto 0\}, \ldots, e\{i \mapsto n-1\}$. As the informal typing rules in Figure 2 show, the difference between ary and Plate is that ary is non-probabilistic: it neither requires nor produces a distribution (like `Data.Vector.generate` in Haskell). Hence $\mathsf{ary}(m, i, e)[e']$ reduces to $e\{i \mapsto e'\}$.[2]

Eliminating the latent array variable $\vec{\theta}$ is trickier than eliminating $\vec{x}$. Because most elements of $\vec{\theta}$ use multiple elements of $\vec{p}$, we cannot eliminate $\vec{\theta}$ just by reordering the iterations of the loop where $\vec{\theta}$ is used (step 3 above). Rather, we need to work with mathematical denotations underlying the source program. Eliminating $\vec{\theta}$ amounts to performing the $(m-1)$-dimensional integral

$$\int_{\mathbb{R}^{m-1}} \underbrace{\left(\prod_{i=0}^{m-2} \vec{p}[i]^{\alpha(i)} (1-\vec{p}[i])^{\beta(i)}\right)}_{\text{step 1}} \underbrace{\left(\prod_{j=0}^{n-1} \left(\textstyle\prod_{k=0}^{\vec{y}[j]-1} \vec{p}[k]\right) \left\{\begin{smallmatrix} 1-\vec{p}[\vec{y}[j]] & \vec{y}[j]<m-1 \\ 1 & \vec{y}[j]=m-1 \end{smallmatrix}\right.\right)}_{\text{step 3}} \underbrace{\left(\textstyle\prod_{k=0}^{z-1} \vec{p}[k]\right) \left\{\begin{smallmatrix} 1-\vec{p}[z] & z<m-1 \\ 1 & z=m-1 \end{smallmatrix}\right.}_{\text{step 5}} d\vec{p}. \quad (13)$$

The factors in the integrand arise from steps 1, 3, and 5 above and the macro expansion (12). The way to calculate this integral is to group the factors $\vec{p}[k]$, $1-\vec{p}[\vec{y}[j]]$, and $1-\vec{p}[z]$ by which elements of $\vec{p}$ they use. For instance, the factors $\prod_{k=0}^{\vec{y}[j]-1} \vec{p}[k]$ include $\vec{p}[i]$ at exactly those j for which $i < \vec{y}[j]$. Thus, the integral (13) can be rewritten to the form

$$\int_{\mathbb{R}^{m-1}} \left(\prod_{i=0}^{m-2} \vec{p}[i]^{\alpha'(i)} (1-\vec{p}[i])^{\beta'(i)}\right) d\vec{p} = \prod_{i=0}^{m-2} \int_{\mathbb{R}} \vec{p}[i]^{\alpha'(i)} (1-\vec{p}[i])^{\beta'(i)} d\vec{p}[i], \quad (14)$$

whose exponents

$$\alpha'(i) = \underbrace{\alpha(i)}_{\text{step 1}} + \underbrace{\left(\sum_{j=0}^{n-1} \left\{\begin{smallmatrix} 1 & i<\vec{y}[j] \\ 0 & \text{otherwise} \end{smallmatrix}\right.\right)}_{\text{step 3}} + \underbrace{\left\{\begin{smallmatrix} 1 & i<z \\ 0 & \text{otherwise} \end{smallmatrix}\right.}_{\text{step 5}} \qquad \beta'(i) = \underbrace{\beta(i)}_{\text{step 1}} + \underbrace{\left(\sum_{j=0}^{n-1} \left\{\begin{smallmatrix} 1 & i=\vec{y}[j]<m-1 \\ 0 & \text{otherwise} \end{smallmatrix}\right.\right)}_{\text{step 3}} + \underbrace{\left\{\begin{smallmatrix} 1 & i=z<m-1 \\ 0 & \text{otherwise} \end{smallmatrix}\right.}_{\text{step 5}} \quad (15)$$

[1] We use $m-1$ Beta distributions, not m Gamma distributions, even though normalizing an array of m independent Gamma variables is another well-known way to obtain the Dirichlet distribution. The reason is that every element of the normalized array depends on every element of the unnormalized array, so this more symmetric way to obtain the Dirichlet distribution actually makes it harder to eliminate $\vec{\theta}$ and to recognize the conjugacy of $\vec{y}$ and z.

[2] We leave the meaning of indexing out of bounds undefined.

have absorbed terms from steps 3 and 5.[3] The right-hand side of (14) is a product of *independent* one-dimensional integrals that existing computer algebra can finally calculate.[4]

In sum, simplifying an adequate variety of array programs requires representing high- and arbitrary-dimensional integrals and uncovering independence among the dimensions that is not necessarily expressible at the source level. We flesh out this approach below. It succeeds on all the examples above.

3.3 High- and arbitrary-dimensional integrals

Our simplifier handles arrays by converting them to high- and arbitrary-dimensional integrals. It takes the same three steps as Carette and Shan's scalar simplifier [2016]. We illustrate these steps using the relatively simple example (9) above. First, our simplifier converts (9) into the expression

$$\int_{\mathbb{R}^n}\left(\prod_{i=0}^{n-1}\frac{\mathrm{e}^{-\frac{1}{2}(\vec{x}[i]-\mu)^2}}{\sqrt{2\pi}}\right)\int_{\mathbb{R}^n}\left(\prod_{i=0}^{n-1}\frac{\mathrm{e}^{-\frac{1}{2}(\vec{y}[i]-\vec{x}[i])^2}}{\sqrt{2\pi}}\right)\int_{\mathbb{R}^n}\left(\prod_{i=0}^{n-1}\frac{\mathrm{e}^{-\frac{1}{2}(\vec{z}[i]-\vec{x}[i])^2}}{\sqrt{2\pi}}\right)h([\vec{y},\vec{z}])\,d\vec{z}\,d\vec{y}\,d\vec{x}. \tag{16}$$

Second, it integrates over the latent variable $\vec{x}$ to get

$$\int_{\mathbb{R}^n}\int_{\mathbb{R}^n}2^{-n}3^{-\frac{1}{2}n}\pi^{-n}\mathrm{e}^{-\frac{1}{3}n\mu^2}\mathrm{e}^{-\frac{1}{3}\sum_{i=0}^{n-1}\vec{y}[i]^2}\mathrm{e}^{\frac{1}{3}\mu\sum_{i=0}^{n-1}\vec{y}[i]}\mathrm{e}^{-\frac{1}{3}\sum_{i=0}^{n-1}\vec{z}[i]^2}\mathrm{e}^{\frac{1}{3}\mu\sum_{i=0}^{n-1}\vec{z}[i]}\mathrm{e}^{\frac{1}{3}\sum_{i=0}^{n-1}\vec{y}[i]\vec{z}[i]}h([\vec{y},\vec{z}])\,d\vec{z}\,d\vec{y}. \tag{17}$$

Third, it converts this expression back to the program (10).

Although conceptually straightforward, extending these three steps to handle arrays is challenging because computer algebra systems today only support integrals whose dimensionality is low and known, not high and arbitrary. Even just to represent the integrals—let alone compute with them—we had to extend the language of expressions.

Our representation for high- and arbitrary-dimensional integrals appears at the end of Figure 3:

$$g ::= \cdots \mid \int_X g\,d\vec{\boldsymbol{x}} \qquad\qquad X ::= (a,b) \mid \prod_{i=c}^{d} X \tag{18}$$

Whereas in $\int_a^b g\,dx$ the variable x ranges over reals, in $\int_X g\,d\vec{\boldsymbol{x}}$ the variable $\vec{\boldsymbol{x}}$ ranges over arrays of (arrays of …) reals. The space X is either a real interval (a,b) or a Cartesian product $\prod_{i=c}^{d} Y(i)$ indexed by integers i between c and d. For example, $\int_a^b f(x)\,dx$ is equivalent to $\int_{(a,b)} f(x)\,dx$, and $\int_{a_0}^{b_0}\int_{a_1}^{b_1}\int_{a_2}^{b_2} f([x,y,z])\,dz\,dy\,dx$ is equivalent to $\int_{\prod_{i=0}^{2}(a_i,b_i)} f(\vec{x})\,d\vec{x}$.

In the integral $\int_X g\,d\vec{\boldsymbol{x}}$ over the space $X = \prod_{i_1=c_1}^{d_1}\cdots\prod_{i_r=c_r}^{d_r}(a,b)$, the set of valid indices into the array $\vec{\boldsymbol{x}}$ is determined by the sequence of index-variable bindings $[{}_{i_1=c_1}^{d_1}, \ldots, {}_{i_r=c_r}^{d_r}]$. We notate this sequence of name-bounds pairs by the metavariable B, then define the syntactic sugar

$$\textstyle\prod_B X = \prod_{i_1=c_1}^{d_1}\cdots\prod_{i_r=c_r}^{d_r} X, \quad \mathrm{ary}(B,e) = \mathrm{ary}(d_1-c_1+1, i_1, \ldots \mathrm{ary}(d_r-c_r+1, i_r, e)\ldots), \tag{19}$$

$$\textstyle\prod_B e = \prod_{i_1=c_1}^{d_1}\cdots\prod_{i_r=c_r}^{d_r} e, \qquad e[B] = e[i_1+c_1]\ldots[i_r+c_r]. \tag{20}$$

Our new first step is defined using this new notation. Figure 4 shows the key cases. In the call integrate(m, B, h), the second argument B is a new accumulator that tracks the Plate levels nested around m. This list starts empty, and grows when integrate encounters Plate. When integrate arrives at a primitive distribution such as Gaussian, it generates an integral whose body nests as many definite products as the list is long.

[3] This absorption can also be viewed as the conjugacy of binomial likelihoods with respect to Beta distributions.

[4] Expressing a multivariate distribution by transforming an array of independent random variables is a general strategy that may apply beyond Dirichlet distributions. For example, it is promising to express a multivariate Gaussian distribution by transforming an array of independent one-dimensional Gaussian random variables, but we have only tried it (successfully) with known (non-diagonal) covariance matrices.

$$
\begin{aligned}
\mathsf{integrate}\big(\mathsf{Gaussian}(\mu,\sigma), B, h\big) &= \int_{\prod_B(-\infty,\infty)} \Big(\prod\nolimits_B \frac{\mathrm{e}^{-\frac{1}{2\sigma^2}(\vec{\boldsymbol{x}}[B]-\mu)^2}}{\sqrt{2\pi}\sigma}\Big)\, h(\vec{\boldsymbol{x}})\, d\vec{\boldsymbol{x}} \\
\mathsf{integrate}\big(\mathsf{Ret}(e), B, h\big) &= h\big(\mathsf{ary}(B, e)\big) \\
\mathsf{integrate}\big(\mathsf{Weight}(e, m), B, h\big) &= \Big(\prod\nolimits_B e\Big) \cdot \mathsf{integrate}(m, B, h) \\
\mathsf{integrate}\big(\mathsf{Bind}(m, x, m'), B, h\big) &= \mathsf{integrate}\big(m, B, \lambda\vec{\boldsymbol{x}}.\, \mathsf{integrate}(m'\{x \mapsto \vec{\boldsymbol{x}}[B]\}, B, h)\big) \\
\mathsf{integrate}\big(\mathsf{Plate}(e, j, m), B, h\big) &= \mathsf{integrate}(m, [B, {}^{e-1}_{j=0}], h)
\end{aligned}
$$

Fig. 4. Converting programs with arrays to patently linear expressions

Our second step seeks to eliminate latent array variables by integrating over them. In (16) for example, we seek to integrate $\int_{\mathbb{R}^n}(\prod_i \cdots)(\prod_i \cdots)(\prod_i \cdots)\, d\vec{x}$ symbolically. We perform such an integral by factoring it into a product of *independent* one-dimensional integrals. Formally, suppose we want to perform an integral $\int_X f(\vec{\boldsymbol{t}})\, d\vec{\boldsymbol{t}}$ over the space $X = \prod_B(a, b)$. We try to re-express its body $f(\vec{\boldsymbol{t}})$ as

$$e_0 \cdot \prod_B g(\vec{\boldsymbol{t}}[B]), \tag{21}$$

where g depends on just one element of $\vec{\boldsymbol{t}}$ at a time. If this rewrite succeeds, then the integral factors into a product of one-dimensional integrals over a scalar variable t:

$$\int_X f(\vec{\boldsymbol{t}})\, d\vec{\boldsymbol{t}} = \int_X e_0 \cdot \prod_B g(\vec{\boldsymbol{t}}[B])\, d\vec{\boldsymbol{t}} = e_0 \cdot \prod_B \int_a^b g(t)\, dt \tag{22}$$

In our running example, the array case reduces to the scalar case of integrating over x in (6):

$$\int_{\mathbb{R}^n} \prod_{i=0}^{n-1} \frac{\mathrm{e}^{-\frac{1}{2}(\vec{x}[i]-\mu)^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(\vec{y}[i]-\vec{x}[i])^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(\vec{z}[i]-\vec{x}[i])^2}}{\sqrt{2\pi}} d\vec{x} = \prod_{i=0}^{n-1} \int_{\mathbb{R}} \frac{\mathrm{e}^{-\frac{1}{2}(t-\mu)^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(\vec{y}[i]-t)^2}}{\sqrt{2\pi}} \frac{\mathrm{e}^{-\frac{1}{2}(\vec{z}[i]-t)^2}}{\sqrt{2\pi}} dt \tag{23}$$

Existing routines for integrals and definite products then directly apply to eliminate the latent $\vec{x}$, even if n were unknown. (If rewriting to (21) fails, then the latent variable would not be eliminated.)

To recognize array distributions, the third step tries to rewrite a density $f(\vec{\boldsymbol{t}})$ to a product (21). If this succeeds and the resulting factor g is the density of some one-dimensional distribution m, then f is the density of r levels of Plate nested around m. Continuing the example, the right-hand-side of (23) is already a product whose body depends on just one element of $\vec{z}$ at a time, so again the array case reduces to the scalar case (5), and our simplifier recognizes (23) to be the density of $\mathsf{Plate}(n, i, \mathsf{Gaussian}(\frac{1}{2}(\mu + \vec{y}[i]), \frac{\sqrt{6}}{2}))$ at $\vec{z}$.

3.4 Rewriting an expression as a product

The purpose of the *unproduct* operation is to rewrite an expression as a product (21). As just described, this rewrite is key to eliminating array variables and recognizing array distributions in the second and third steps of our extended simplifier. Because the running example (9) above is simple, the unproduct rewrite to (23) is trivial. It turns out that we can handle a much broader variety of array programs that express desired algorithms—including all the examples in Section 3.2—by making the unproduct operation succeed more often.

The unproduct operation enables the automation of many common simplifications, by uncovering independence among random variables and likelihood factors that is prevalent yet often hidden in the source program. It generalizes the *normalization* rewrite rule for indirect indexing in AugurV2 [Huang et al. 2017], as illustrated by the Gaussian mixture model in Section 3.2. It also generalizes *inversion* in the lifted inference literature [de Salvo Braz and O'Reilly 2017] from discrete distributions to continuous ones. At the very least, because the unproduct operation is

the only way for our extended simplifier to produce Plate, it must succeed in order for a program containing Plate to even just simplify to itself unscathed. (Our test suite has many such *round-trip* tests.) Hence, unproduct needs to succeed even though factors tend to have their parts shuffled by computer algebra. In particular, because our simplifier rewrites $\prod \mathrm{e}^{\cdots}$ to $\mathrm{e}^{\sum \cdots}$ so as to expose holonomy, the two forms need to be treated equivalently by the unproduct operation.

More formally, given a term e and an array variable $\vec{x}$, the goal of the operation unproduct$(e, \vec{x})$ is to produce a pair of expressions (e', g) such that g does not contain $\vec{x}$ free yet $e = e' \cdot \prod_i g(i, \vec{x}[i])$. Because the produced factor g does not contain $\vec{x}$ free but rather takes an index i and an element $\vec{x}[i]$ as inputs, it only gets to use one element of $\vec{x}$ at a time. We call this operation unproduct because its specification is that putting $\prod_i$ on its output should be equal to its input.

The unproduct operation proceeds by structural recursion over a term, remembering the path to the subterm currently in focus. We represent the path as a *heap*. It is a context—an expression with a single hole $[\,]$ where a subexpression can be plugged in. The result of plugging an expression e into a heap H is notated $H[e]$. We distinguish between heaps of two *modes* by what they *factor over*: $H^{\times}$ of mode $\times$ factors over multiplication, whereas H^{+} of mode + factors over addition. For example, $H^{\times}$ could be $[\,]^2$ because $(e_1 \cdot e_2)^2 = e_1^2 \cdot e_2^2$, whereas H^{+} could be $\mathrm{e}^{[\,]}$ because $\mathrm{e}^{e_1+e_2} = \mathrm{e}^{e_1} \cdot \mathrm{e}^{e_2}$. More generally, we maintain the factoring invariants

$$H^{\times}[1] = 1 \qquad H^{\times}[e_1 \cdot e_2] = H^{\times}[e_1] \cdot H^{\times}[e_2] \qquad H^{\times}\left[\textstyle\prod_{i=a}^{b} e\right] = \prod_{i=a}^{b} H^{\times}[e] \tag{24}$$

$$H^{+}[0] = 1 \qquad H^{+}[e_1 + e_2] = H^{+}[e_1] \cdot H^{+}[e_2] \qquad H^{+}\left[\textstyle\sum_{i=a}^{b} e\right] = \prod_{i=a}^{b} H^{\times}[e] \tag{25}$$

by defining a restricted grammar of heaps

$$H^{\times} ::= [\,] \;\big|\; H^{\times}\left[[\,]^{c}\right] \;\big|\; H^{\times}\left[\textstyle\prod_{i=a}^{b}[\,]\right] \;\big|\; H^{\times}\left[\begin{cases}[\,] & e \\ 1 & \text{otherwise}\end{cases}\right] \tag{26}$$

$$H^{+} ::= H^{\times}\left[c^{[\,]}\right] \;\big|\; H^{+}\left[c \cdot [\,]\right] \;\big|\; H^{+}\left[\textstyle\sum_{i=a}^{b}[\,]\right] \;\big|\; H^{+}\left[\begin{cases}[\,] & e \\ 0 & \text{otherwise}\end{cases}\right] \tag{27}$$

where the expressions a, b, c are constants in the sense that they do not contain $\vec{x}$ free. An occurrence of $\prod_{i=a}^{b}$ or $\sum_{i=a}^{b}$ in a heap binds the index variable i.

The goal of unproduct$(e, \vec{x}, H)$, where the accumulator argument H is initially the empty heap $[\,]$, is to produce a pair of expressions (e', g) such that g does not contain $\vec{x}$ free yet $H[e] = e' \cdot \prod_i g(i, \vec{x}[i])$. Again, because g does not contain $\vec{x}$ free but rather takes i and $\vec{x}[i]$ as inputs, it only gets to use one element of $\vec{x}$ at a time.

The definition of unproduct appears in Figure 5. The notation $e' \;?\; e$ means the conditional $\begin{cases} e & e' \\ I & \text{otherwise}\end{cases}$ where I is the identity of the mode of the surrounding heap. That is, we define

$$H^{\times}[e' \;?\; e] = H^{\times}\left[\begin{cases} e & e' \\ 1 & \text{otherwise}\end{cases}\right], \qquad H^{+}[e' \;?\; e] = H^{+}\left[\begin{cases} e & e' \\ 0 & \text{otherwise}\end{cases}\right]. \tag{28}$$

The second case in Figure 5 is the workhorse; it is the source of any g returned that is not just constantly 1. It applies when there is a unique index a where the term e uses the array $\vec{x}$. It would then be correct to return

$$\big(1, \lambda(i, xi).\, H[(i = a) \;?\; e(xi)]\big). \tag{29}$$

For example, unproduct can rewrite $f(\vec{x}[k])$ to $\prod_{i=0}^{n-1} \begin{cases} f(\vec{x}[i]) & i=k \\ 1 & \text{otherwise}\end{cases}$. However, to prevent subsequent computer algebra from stumbling over the conditional, we further reduce the result (29) algebraically in four steps.

(1) Given $H[e' \,?\, e]$, hoist the test e' as far out of H as possible: First, decompose H into $H = H_1[C]$, where the context C is the maximal inner portion of H that does not bind any free variable in e'. (In particular, if H does not bind any free variable in e' at all, then $H_1 = [\,]$ and $C = H$. Otherwise, H_1 has the form $H_1^{\times}\left[\prod_j [\,]\right]$ or $H_1^{+}\left[\sum_j [\,]\right]$, where j is the innermost-scoped index variable bound by H that e' contains free.) Then, rewrite $H_1\left[C[e' \;?\; e]\right]$ to $H_1\left[e' \;?\; C[e]\right]$. (The

$$
\begin{array}{lll}
\text{unproduct}(e, \quad \vec{x}, H) & = (H[e], \lambda(i, xi).\ 1) & \text{if } e \text{ does not contain } \vec{x} \text{ free} \\
\text{unproduct}(e(\vec{x}[a]), \vec{x}, H) & = (1, \lambda(i, xi).\ H[(i = a)\ ?\ e(xi)]) & \text{if } e \text{ only uses } \vec{x} \text{ at index } a \\
\text{unproduct}(c^e, \quad \vec{x}, H^\times) & = \text{unproduct}(e, \vec{x}, H^\times[c^{[\ ]}]) & \text{where } c \text{ does not contain } \vec{x} \text{ free} \\
\text{unproduct}(e^c, \quad \vec{x}, H^\times) & = \text{unproduct}(e, \vec{x}, H^\times[[\ ]^c]) & \text{where } c \text{ does not contain } \vec{x} \text{ free} \\
\text{unproduct}(c \cdot e, \quad \vec{x}, H^+) & = \text{unproduct}(e, \vec{x}, H^+[c \cdot [\ ]]) & \text{where } c \text{ does not contain } \vec{x} \text{ free} \\
\text{unproduct}(\prod_{i=a}^{b} e, \vec{x}, H^\times) & = \text{unproduct}(e, \vec{x}, H^\times[\prod_{i=a}^{b}[\ ]]) & \text{where } a, b \text{ do not contain } \vec{x} \text{ free} \\
\text{unproduct}(\sum_{i=a}^{b} e, \vec{x}, H^+) & = \text{unproduct}(e, \vec{x}, H^+[\sum_{i=a}^{b}[\ ]]) & \text{where } a, b \text{ do not contain } \vec{x} \text{ free} \\
\text{unproduct}(\left\{\begin{smallmatrix} e_1 & d_1 \\ e_2 & d_2 \end{smallmatrix}\right., \vec{x}, H) & = (e_1' \cdot e_2', g_1 \odot g_2) & \text{where } (e_k', g_k) = \text{unproduct}(e_k, \vec{x}, H[d_k\ ?\ [\ ]]) \\
\text{unproduct}(e_1 \cdot e_2, \vec{x}, H^\times) & = (e_1' \cdot e_2', g_1 \odot g_2) & \text{where } (e_k', g_k) = \text{unproduct}(e_k, \vec{x}, H^\times) \\
\text{unproduct}(e_1 + e_2, \vec{x}, H^+) & = (e_1' \cdot e_2', g_1 \odot g_2) & \text{where } (e_k', g_k) = \text{unproduct}(e_k, \vec{x}, H^+) \\
\text{unproduct}(e, \quad \vec{x}, H) & = (H[e], \lambda(i, xi).\ 1) & \text{as the last resort}
\end{array}
$$

Fig. 5. Rewriting an expression as a product: if $\text{unproduct}(e, \vec{x}, H) = (e', g)$, then g does not contain $\vec{x}$ free, yet $H[e] = e' \cdot \prod_i g(i, \vec{x}[i])$. These rules are applied top-down. The first two cases and the last case are the base cases; see the text for algebraic reductions that take place in the second case. The rest are the recursive cases, which simply traverse the structure of the input term e while accumulating the heap H using distributivity. In the last three recursive cases, k is 1 or 2, and $g_1 \odot g_2$ is short for the pointwise product $\lambda(i, xi).\ g_1(i, xi) \cdot g_2(i, xi)$.

identity in $e'\ ?\ e$ may differ from the identity in $e'\ ?\ C[e]$, because the mode of $H_1[C]$ may differ from the mode of H_1.)

(2) Try to turn a loop into a let: Given the loop $\prod_j \left\{\begin{smallmatrix} e(j) & e'(j) \\ 1 & \text{otherwise} \end{smallmatrix}\right.$ or $\sum_j \left\{\begin{smallmatrix} e(j) & e'(j) \\ 0 & \text{otherwise} \end{smallmatrix}\right.$, where the test $e'(j)$ contains j free, try to solve for j in $e'(j)$ to yield an equivalent equation $j = b$. If the solving succeeds, then rewrite the loop to $e(b)$. For example, unproduct can rewrite $\prod_{j=1}^{n} f(j, \vec{x}[j-1])$ to $\prod_{i=0}^{n-1} f(i+1, \vec{x}[i])$, by solving the test $i = j - 1$ symbolically to yield the equivalent equation $j = i + 1$. A more substantial example is that unproduct enables eliminating a Dirichlet distribution by rewriting (13) to (14). However, if the test is $i = \vec{y}[j]$, as in the mixture-model example (11), then the solving fails and this step does nothing.

(3) Try to turn $\prod$ into $\sum$ by pushing it inward: Given the loop $\prod_j \left\{\begin{smallmatrix} e(j) & e'(j) \\ 1 & \text{otherwise} \end{smallmatrix}\right.$ (or just $\prod_j e(j)$), if the body $e(j)$ has the form $e_0^{e_1(j)}$ (or just e_0) where e_0 does not contain j free, then rewrite the loop to $e_0^{\sum_j \left\{\begin{smallmatrix} e_1(j) & e'(j) \\ 0 & \text{otherwise} \end{smallmatrix}\right.}$. If the body $e(j)$ consists of several factors multiplied together, then deal with each factor separately. Similarly, rewrite any factor $\left\{\begin{smallmatrix} e_0^{e_1} & e' \\ 1 & \text{otherwise} \end{smallmatrix}\right.$ at the top level, where e_0 is closed, to $e_0^{e_1 \cdot \left\{\begin{smallmatrix} 1 & e' \\ 0 & \text{otherwise} \end{smallmatrix}\right.}$.

(4) Try to push $\sum$ inward: Given the loop $\sum_j \left\{\begin{smallmatrix} e(j) & e'(j) \\ 0 & \text{otherwise} \end{smallmatrix}\right.$ (or just $\sum_j e(j)$), if the body $e(j)$ has the form $e_0 \cdot e_1(j)$ (or just e_0) where e_0 does not contain j free, then rewrite the loop to $e_0 \cdot \sum_j \left\{\begin{smallmatrix} e_1(j) & e'(j) \\ 0 & \text{otherwise} \end{smallmatrix}\right.$. If the body $e(j)$ expands to several terms added together, then deal with each term separately.

Roughly, these steps work together to reduce the conditional expressions produced by unproduct, so that subsequent computer algebra successfully eliminates latent variables and recognizes primitive distributions in Section 3.2 and our classification benchmarks. These benchmarks use indexing heavily to express clusters, topics, and Dirichlet distributions. For example, unproduct produces the expression on the *right*-hand-side of (11), which is expanded so that the sums and conditionals

do not contain the integration variable x free; existing computer algebra can thus perform the integral $\int_{\mathbb{R}} e^{f(x)}\,dx$ automatically by treating those sums and conditionals atomically.

The last case in Figure 5 is the fallback for when e uses $\vec{x}$ at multiple indices but cannot be decomposed recursively. This fallback is the source of any e' returned that uses $\vec{x}$ as a whole. Because it trivially satisfies the equational specification $H[e] = e' \cdot \prod_i g(i, \vec{x}[i])$, simplification still preserves semantics, but does not eliminate a variable or recognize a conjugacy. In some cases, this is because there is really nothing to do; in other cases, our optimizing compiler lags behind the mathematical prowess of applied statisticians.

4 THE HISTOGRAM TRANSFORMATION

We introduce the *histogram* transformation, which improves the asymptotic running time of loops that arise from simplifying mixture models, by rewriting loops into map-reduce expressions.

Recall that the goal of our compilation pipeline is the efficient execution of array inference algorithms expressed as probabilistic programs denoting conditional distributions. Simplifying these programs produces loops, such as

$$\sum_{j=0}^{n-1} \begin{cases} \vec{s}[j] & i{=}\vec{y}[j] \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

and other summations in (1) (same as in the right-hand-side of equation (11)). When the program performs indirect indexing, the resulting loops are nested: the outer loop iterates over classes i and the inner loop iterates over all individuals j but only considers those that belong to the current class ($i = \vec{y}[j]$). By generalizing loops from scalar summation to other map-reduce expressions, we can dramatically speed up such nested loops to run in time independent of the number of classes. For example, by looking up the class of every individual, a single pass over the population can produce the sum of every class; a summation such as (30) can be computed for all i in $O(n)$ rather than $O(mn)$ time.

Loop nests like (30) often arise for inference when the model divides array elements into subpopulations, as mixture models do. Eliminating latent variables proliferates such loop nests, because intuitively, after eliminating a variable x, the information that used to be required to infer x becomes required to infer the variables that depend on x. For example, in the Gaussian mixture model in Section 3.2, each point $\vec{s}[j]$ only requires one quantity—the mean $\vec{x}[\vec{y}[j]]$ underlying the class $\vec{y}[j]$—but eliminating $\vec{x}$ makes the point $\vec{s}[j]$ require the other points in the same class.

Wherever a nested formula arises, an applied statistician would translate it manually to unnested code as a matter of course; we automate this asymptotic improvement here. As Figure 1 suggests, this histogram optimization of ours composes with simplification and applies to both exact and approximate inference procedures. In fact, it applies to probabilistic and non-probabilistic programs alike, even though probabilistic programming is the context where we needed it and invented it. This modularity and generality sets our work apart from other systems that incorporate this optimization only for MCMC inference on mixture models [Huang et al. 2017; Tristan et al. 2014].

As the name implies, the histogram transformation recognizes nested loops that are usually visualized as (generalized) histograms. These histogram computations manifest as sums such as (30). We thus introduce a term construct Hist to represent such computations. The transformation rewrites such sums to an equivalent let-expression that binds a Hist term to a *hist* variable. For example, in the scope of $i \in \{0, \ldots, m-1\}$, the histogram transformation rewrites (30) to

$$\text{let } hist = \mathsf{Hist}_{j=0}^{n-1}\left(\mathsf{Idx}_i^m(\vec{y}[j], \mathsf{Add}(\vec{s}[j]))\right) \text{ in } hist[i], \tag{31}$$

where the capitalized keywords are new (in Figure 6). The *hist* variable is bound to an array whose size is m and whose element at each index i is the sum of those $\vec{s}$ whose corresponding $\vec{y}$ matches i. The sequential code we generate for computing *hist* initializes it to an all-zero mutable array then

Reducers

$$\frac{\begin{array}{c}[j:\mathbb{N}]\\ \vdots\\ e:\mathbb{R}\end{array}}{\mathsf{Add}(e) \triangleright_j \mathbb{R}} \qquad \frac{\begin{array}{ccc}[j:\mathbb{N}] & [i:\mathbb{N}] & \\ \vdots & \vdots & \\ b:\mathbb{N} & e:\mathbb{N} & r \triangleright_j T\end{array}}{\mathsf{Idx}_i^b(e,r) \triangleright_j \mathbb{A}\, T} \qquad \frac{\begin{array}{cccc}[j:\mathbb{N}] & & & \\ \vdots & & & \\ e:\mathbb{B} & r_1 \triangleright_j T_1 & r_2 \triangleright_j T_2 & \end{array}}{\mathsf{Split}(e,r_1,r_2) \triangleright_j T_1 \times T_2} \qquad \frac{r_1 \triangleright_j T_1 \quad r_2 \triangleright_j T_2}{\mathsf{Fanout}(r_1,r_2) \triangleright_j T_1 \times T_2} \qquad \frac{}{\mathsf{Nop} \triangleright_j \mathbb{1}}$$

Histograms

$$\frac{a:\mathbb{N} \quad b:\mathbb{N} \quad r \triangleright_j T}{\mathsf{Hist}_{j=a}^b(r) : T}$$

Fig. 6. Typing rules for reducer expressions and the histogram expressions they constitute

adds $\vec{s}[j]$ to $hist[\vec{y}[j]]$ for each j from 0 to $n-1$. Roughly, Add means to add, Idx_i^m means to index into an array of size m, and $\mathsf{Hist}_{j=0}^{n-1}$ means to loop for j from 0 to $n-1$. We leave further speedups of such map-reduce computations using parallelization, vectorization, and GPUs to future work.

Out of context, the let-expression (31) seems like a waste because it computes *hist* then uses only one element of it. But because the class variable i does not occur free in the Hist expression (the subscript i is a binder), LICM (Section 5.1) will later lift the binding of *hist* out of the scope of i, thus reusing it across all m classes. To pave the way, a Hist term should depend on as few inner-scoped variables as possible, and the index variable i in $hist[i]$ should be loop-bound.

4.1 Syntax and semantics of reducers

Figure 6 formalizes the sublanguage of reducers, which constitute the body of a Hist expression. The judgment $r \triangleright_j T$ means that r is a reducer of type T over index j. A reducer r constitutes the body of a histogram expression $\mathsf{Hist}_{j=a}^b(r)$, whose typing rule is shown at the bottom of the figure. The scope of the variable j is special, because the histogram expression $\mathsf{Hist}_{j=a}^b(r)$ interprets the reducer r in two ways: first to initialize a mutable histogram to zero independently of j, and then to update the histogram iteratively by looping over the index j. Thus, j can only appear free in certain parts of r, marked intentionally by "$[j:\mathbb{N}]\ldots$" in Figure 6.

Mathematically, a reducer r of type T denotes a monoid whose carrier is T (that is, an associative binary operation $+_r$ on T that has an identity r^0), along with a map r^1 from indices j to elements of T. Intuitively, the histogram expression $\mathsf{Hist}_{j=a}^b(r)$ first initializes a mutable histogram using the identity r^0, then updates the histogram iteratively using the map r^1 and the monoid operation $+_r$. That is, $\mathsf{Hist}_{j=a}^b(r)$ denotes the monoidal sum $r^1(a) +_r \cdots +_r r^1(b)$ (which equals r^0 in case $a = b+1$). Thus, to describe the operational semantics of a histogram expression on sequential hardware, we associate with each reducer r two methods: initializing a mutable T, and updating it at a given index j. The expression $\mathsf{Hist}_{j=a}^b(r)$ uses r to initialize a mutable histogram T then updates it at each index $j = a, \ldots, b$. We now describe the denotation and operation of each reducer construct in turn.

- $\mathsf{Add}(e)$ denotes addition on $\mathbb{R}$ along with the map $\lambda j.\, e$. Accordingly, $\mathsf{Add}(e)$ initializes a real to 0 and updates it by adding e.
- $\mathsf{Idx}_i^b(e, r(i))$ denotes the product of the monoids denoted by $r(0), \ldots, r(b-1)$, along with the map
$$\mathsf{Idx}_i^b(e,r(i))^1 = \lambda j.\, \mathrm{ary}\left(b, i, \begin{cases} r(e)^1(j) & i{=}e \\ r(i)^0 & \text{otherwise}\end{cases}\right). \tag{32}$$
Accordingly, $\mathsf{Idx}_i^b(e, r(i))$ initializes an array of size b by initializing its elements using $r(0), \ldots, r(b-1)$, and updates the array by updating just the element at e using $r(e)$.
Note that the denoted monoid and the initialization method are independent of e and thus independent of j. In particular, the size expression b is evaluated during initialization without

$$\text{histogram}\big(C\big[\begin{cases} e_1 & e \\ e_2 & \text{otherwise}\end{cases}\big], j\big) = \Big(\text{Fanout}(m_1, m_2), \lambda(s_1, s_2).\ \begin{cases} f_1(s_1) & e \\ f_2(s_2) & \text{otherwise}\end{cases}\Big)$$
$$\text{where } (m_k, f_k) = \text{histogram}(C[e_k], j) \text{ and } e \text{ does not depend on } j$$
$$\text{histogram}\big(C\big[\begin{cases} e_1 & e \\ e_2 & \text{otherwise}\end{cases}\big], j\big) = \big(\text{Split}(e, m_1, m_2), \lambda(s_1, s_2).\ f_1(s_1) + f_2(s_2)\big)$$
$$\text{where } (m_k, f_k) = \text{histogram}(C[e_k], j)$$
$$\text{histogram}\Big(\begin{cases} a & i{=}e \\ 0 & \text{otherwise}\end{cases}, \quad j\Big) = \Big(\text{Idx}_i^m(e, r), \lambda s.\ \begin{cases} f(s[i]) & i \in \{0, \ldots, m-1\} \\ 0 & \text{otherwise}\end{cases}\Big)$$
$$\text{where } (r, f) = \text{histogram}(a, j),\ i \text{ is a loop-bound variable that does not depend on } j,$$
$$\text{and the context entails that } i \in \{0, \ldots, m-1\} \text{ or } e \in \{0, \ldots, m-1\}$$
$$\text{histogram}\big(0, \quad j\big) = \big(\text{Nop}, \lambda s.\ 0\big)$$
$$\text{histogram}\big(e, \quad j\big) = \big(\text{Add}(e), \lambda s.\ s\big)$$

Fig. 7. Rewriting a summation as a histogram: if $\text{histogram}(e, j) = (r, f)$ then $\sum_{j=0}^{n-1} e = f\big(\text{Hist}_{j=0}^{n-1}(r)\big)$. The metavariable C denotes a context. These rules are applied top-down, except the second and third rules are prioritized by choosing the rule for which the innermost scope of the free variables $FV(e) \setminus \{j\}$ is outermost.

using j. However, Idx_i^b binds i in $r(i)$, so the monoid and initialization of each histogram element $hist[i]$ can depend on i. In particular, $r(i)$ may contain an inner $\text{Idx}_{i'}^{b'}$ whose size expression b' does depend on i (not j). Such a nested Idx reducer produces a histogram that is a ragged array of arrays. We cannot forego the bound variable i by substituting e for i in b', because e may contain j free.

- $\text{Split}(e, r_1, r_2)$ and $\text{Fanout}(r_1, r_2)$ both denote the product of the monoids denoted by r_1 and r_2. But

$$\text{Split}(e, r_1, r_2)^1 = \lambda j.\ \begin{cases} (r_1^1(j), r_2^0) & e \\ (r_1^0, r_2^1(j)) & \text{otherwise}\end{cases}, \qquad \text{Fanout}(r_1, r_2)^1 = \lambda j.\ \big(r_1^1(j), r_2^1(j)\big). \tag{33}$$

  Accordingly, $\text{Split}(e, r_1, r_2)$ and $\text{Fanout}(r_1, r_2)$ both initialize a pair by initializing its parts using r_1 and r_2. But Split uses r_1 to update the first part when e is true and uses r_2 to update the second part when e is false, whereas Fanout always updates both parts.
- Nop denotes the trivial monoid and the constant map. Accordingly, Nop initializes a unit value and does nothing to it.

4.2 Histogram transformation implementation

We recognize when a $\sum_{j=0}^{n-1} e$ can be rewritten in terms of an equivalent Hist computation that can then be hoisted by LICM for reuse. Formally, we describe a program transformation histogram such that if $\text{histogram}(e, j) = (r, f)$ then $\sum_{j=0}^{n-1} e = f\big(\text{Hist}_{j=0}^{n-1}(r)\big)$. To facilitate LICM, r should depend on as few inner-scoped variables as possible.

The entire definition of histogram appears in Figure 7. Whenever we encounter a summation $\sum_{j=0}^{n-1} e$, we apply the rules in Figure 7 to evaluate $\text{histogram}(e, j)$ to (r, f), then replace $\sum_{j=0}^{n-1} e$ by $f\big(\text{Hist}_{j=0}^{n-1}(r)\big)$ if r looks profitable (that is, contains Idx or Fanout).

The histogram transformation is profitable when the summand chooses among alternatives, typically depending on some contextual information (such as i in (31)). The first rule takes all expressions defined by cases which do not depend on the summation variable j, and translates them to a Fanout. Further case expressions are translated to either a Split or an Idx, by pulling out conditions while prioritizing outermost bound variables. Once all case expressions are gone, the remainder is emitted either as Nop (if zero) or Add.

Continuing with the example (30), we try histogram $\left(\begin{cases}\vec{s}[j] & i=\vec{y}[j] \\ 0 & \text{otherwise}\end{cases}, j\right)$. The first rule does not apply, as the condition $i = \vec{y}[j]$ depends on j. The next two rules are both applicable: the Split rule incurs the free variables $\{i, \vec{y}\}$ whereas the Idx rule only incurs $\{\vec{y}\}$. The Idx rule wins, as the input $\vec{y}$ is bound outside i. We end up with histogram$(\vec{s}[j], j)$, which only matches the last rule. Assembling the results gives $\left(\mathsf{Idx}_i^m(\vec{y}[j], \mathsf{Add}(\vec{s}[j])), \lambda hist.\ hist[i]\right)$ as desired.

5 CODE GENERATION

Our code generator uses the domain specific properties of Hakaru programs to generate optimized x86 code at runtime. This generator is designed to fit into the pipeline of Figure 1—after the programs have undergone the simplification and histogram transformations—although it applies to any Hakaru program. In fact, the optimizations performed by the generator make sense for a general-purpose language (GPL) and are not new, but thanks to the invariants present in Hakaru programs, we can implement them much more easily, perform them much more aggressively, and reap much more performance benefit. And we need to: as we demonstrate in the ablation study in Section 6.2, simplifying array programs that express desired inference algorithms produces residual code—such as repeated traversals of arrays—that would be prohibitively slow without optimization.

The time-consuming computations of probabilistic programs come from pure numerical expressions involving tuples and arrays. It is straightforward to translate these programs into any GPL. However, the domain-specific nature of Hakaru provides several advantages for generating efficient code, advantages not typically available to GPLs:

(1) All arrays in Hakaru programs are immutable and unaliased, and loops operate over arrays.
(2) The histogram transformation produces loops that are nested yet independent.
(3) Hakaru programs not only contain loops but typically *are* the loop body of an inference method, so they are both short and called repeatedly on a particular data set.

Using these insights, the second half of our pipeline (the bottom half of Figure 1) optimizes programs in two ways that are novel in the context of probabilistic programming languages:

(1) We perform LICM [Aho et al. 1986] to hoist inner loops out of outer loops. We then fuse loops of the same bounds together while lowering the program into *Sham IR*, an IR with for loops and mutation that compiles to x86 via LLVM. We carry out these simple optimizations freely and aggressively, without worrying about side effects (Section 5.1). These optimizations yield a 1289× speedup (Table 2).
(2) We JIT-compile Hakaru programs at run time, allowing for extensive specialization (Section 5.2) yielding a 9.5× speedup (Table 2).

Our code generator uses exact arithmetic but generates code that uses floating-point arithmetic. It is well known that floating-point probabilities should be computed in log-space in order to avoid underflow. We use this log-representation for all numbers of type $\mathbb{R}^+$.

5.1 Loop optimizations

LICM and loop fusion are the two most significant optimizations performed by our code generator. As depicted in Figure 1, LICM operates on A-normal forms [Flanagan et al. 1993] in our pure (probabilistic) language, before loop fusion lowers them into Sham's imperative IR. This design makes the optimizations easier to implement and more effective, as we now describe.

The input language to our LICM pass makes it easy to identify loops and compute their dependencies. That is important as we want to find where we can convert a nest of loops into a sequence of loops—that is, when an inner loop does not depend on an outer loop's index variable. Such code motion yields our biggest performance gain, in part due to the preceding histogram transformation. Identifying loops is simple, because Hakaru has only four specialized loop constructs ($\sum$, $\prod$, ary,

$$
\begin{array}{l}
\lambda\vec{\alpha}:\mathbb{ARR}^{+}.\ \lambda\vec{y}:\mathbb{AN}.\ \lambda\vec{s}:\mathbb{AR}.\ \lambda u:\mathbb{N}.\\
\text{let } hist_1 = \mathsf{Hist}_{k=0}^{\#\vec{s}-1}\left(\mathsf{Idx}_{\_}^{\#\vec{\alpha}}(\vec{y}[k], \mathsf{Add}(1))\right)\\
\quad hist_2 = \mathsf{Hist}_{k=0}^{\#\vec{s}-1}\left(\mathsf{Idx}_{\_}^{\#\vec{\alpha}}(\vec{y}[k], \mathsf{Add}(\vec{s}[k]))\right)\\
\text{let } array_1 = \mathsf{ary}(\#\vec{\alpha}, i,\\
\qquad \text{let } prod_1 = \prod_{j=0}^{\#\vec{\alpha}-1}\left(hist_1[j] + \begin{cases}\vec{y}[u] & j=i\\ 0 & \text{otherwise}\end{cases}\right)\\
\qquad\quad sum_1 = \sum_{j=0}^{\#\vec{\alpha}-1}\left(hist_2[j] + \begin{cases}\vec{s}[u] & j=i\\ 0 & \text{otherwise}\end{cases}\right)\\
\qquad prod_1 + sum_1)\\
\mathsf{Categorical}(array_1)
\end{array}
$$

Fig. 8. An excerpt from one of our examples after performing LICM. Here $hist_1$ and $hist_2$ were moved out of the $prod_1$ and sum_1 loops respectively, and out of the $array_1$ loop together.

$$
\begin{array}{l}
\lambda\vec{\alpha}:\mathbb{ARR}^{+}.\ \lambda\vec{y}:\mathbb{AN}.\ \lambda\vec{s}:\mathbb{AR}.\ \lambda u:\mathbb{N}.\\
\text{let } hist_1 := \mathsf{newArray}(\#\vec{\alpha})\\
\quad hist_2 := \mathsf{newArray}(\#\vec{\alpha})\\
\text{for } k = 0 \text{ to } \#\vec{s} - 1:\\
\quad hist_1[\vec{y}[k]] \mathrel{+}= 1;\ hist_2[\vec{y}[k]] \mathrel{+}= \vec{s}[k]\\
\text{let } array_1 := \mathsf{newArray}(\#\vec{\alpha})\\
\text{for } i = 0 \text{ to } \#\vec{\alpha} - 1:\\
\quad \text{let } prod_1 := 1;\ sum_1 := 0\\
\quad \text{for } j = 0 \text{ to } \#\vec{\alpha} - 1:\\
\qquad prod_1 \times\!= hist_1[j] + \begin{cases}\vec{y}[u] & j=i\\ 0 & \text{otherwise}\end{cases}\\
\qquad sum_1 \mathrel{+}= hist_2[j] + \begin{cases}\vec{s}[u] & j=i\\ 0 & \text{otherwise}\end{cases}\\
\quad array_1[i] := prod_1 + sum_1\\
\mathsf{Categorical}(array_1)
\end{array}
$$

Fig. 9. The result of loop fusion and lowering on the example in Figure 8

Hist) and no general recursion. Computing dependencies using A-normalization in a pure language ensures that code motion preserves semantics: we hoist let-bindings as far out as the scope of their free variables allows. Figure 8 shows how a typical program looks like after LICM and before loop fusion and lowering; the two Hist expressions, which were originally nested inside two loops, did not depend on them and have been safely hoisted.

Next, multiple independent loops with identical bounds can be fused. In our domain, aggressive loop fusion improves performance because most loops iterate over arrays and fusion reduces the number of indexing operations. In contrast, loop fusion in a GPL may worsen performance by disturbing locality of reference.

Although Hakaru makes loop fusion straightforward, it is inappropriate as the output language, because a single fused loop may need to maintain many accumulators without tupling them. Instead, our loop-fusion pass produces Sham IR, which has for-loops and mutation. A single pass fuses loops and lowers them to Sham IR, to avoid the harder task of identifying independent loops in Sham IR. Figure 9 shows the result of loop fusion on the example from Figure 8.

Applying LICM and loop fusion to histogram operations introduces multiple array indexing operations that were previously implicit. If two histograms over the same array were fused, the resulting loop body would contain repeated indexing operations, such as $\vec{y}[k]$ in Figure 9. To avoid this repeated indexing, we follow loop fusion by a hoisting pass in Sham IR that applies only to indexing operations into input arrays, which are known to be constant. This helps reduce memory lookup and improve cache locality should these loops be unrolled later.

5.2 Run-time specialization and code generation

Our programs are small and typically run as the body of an outer loop over fixed-size data. To use this fact, we perform several optimizations that can only be performed in a JIT compiler. Inside the outer loop, some information stays the same across iterations; in particular, arrays whose values change may well stay a constant size nevertheless. Thus we allow the programmer to mark arguments with such binding-time information.

When array sizes are known, exact loop bounds tend to become known for most loops. LLVM can then optimize those loops more aggressively. From input array sizes we can even infer intermediate

array sizes. When we know the constant size of an intermediate array, we pre-allocate it only once and reuse it across iterations, removing per-iteration allocation overhead. Thus for array arguments, two different specialization directives can be given: known size, and known size and values.

By waiting until we know array sizes before generating code, we can prepone allocation even further: we can allocate intermediate arrays before we even emit the code! In other words, upon execution of a program, we can use the size of input data to allocate arrays of the appropriate size for intermediate data. The machine code we emit then embeds the intermediate arrays' sizes as well as *addresses* as constants, which no longer need to be kept in registers. We end up with extra registers that can be used for other variables, reducing the need to store and load things on stack.

To perform the run-time specializations as described, we build LLVM IR in memory and JIT-compile it using LLVM's C-API. The outcome, as shown in Section 6, is highly optimized code compared to traditional implementations of domain-specific languages.

6 EVALUATION

The main claim of this paper is that array inference algorithms, expressed as probabilistic programs denoting conditional distributions, can be compiled automatically to efficient code. It is impossible to evaluate how existing systems compile the same programs to implement the same algorithms, because they don't. Instead, we justify our claim by ballpark quantitative comparisons on flagship applications of the decades of work in applied statistics that established the importance of this class of algorithms. We make two overall findings:

- Compared against handwritten code for the same algorithms, we find that Hakaru's generated code achieves competitive speed (and of course the same accuracy).
- Compared against existing systems that use different inference algorithms for the same models, we find that Hakaru delivers the expected increase in accuracy and/or speed.

We measure the performance of both approximate and exact inference algorithms. For approximate inference using Gibbs sampling, we are

- more accurate and 2–12× as fast as JAGS [Plummer 2003], a popular probabilistic-programming system specialized for Gibbs sampling that cannot eliminate latent variables;
- more accurate or faster than STAN [Carpenter et al. 2017], a popular probabilistic-programming system that carries out other inference algorithms and cannot eliminate latent variables;
- 9× as fast as MALLET [McCallum 2002], a popular document-classification package whose handwritten code performs the same computation as our inference procedure; and
- more accurate than AugurV2 [Huang et al. 2017], a recent research system that like Hakaru can compile models with arrays into fast MCMC samplers, but cannot eliminate variables.

For exact inference, we are

- over 5000000× faster while handling 10× more data than PSI [Gehr et al. 2016], another system that can perform exact inference on models containing arrays; and
- 3–11× as fast as handwritten-quality Haskell code emitted by an earlier backend.

All benchmarks were executed on a 6-core AMD-Ryzen 5 with 16 GB of RAM, running Linux 4.15. We used Racket 6.12, LLVM 5.0.1, Maple 2017.2, and GHC 8.0.2.

Our benchmarks span inference tasks that are unsupervised and supervised, with observed and inferred variables that are continuous and discrete. We do not compare against Figaro [Pfeffer 2016] and Anglican [Wood et al. 2014] because those shallowly embedded languages do not use conjugacy to handle unlikely continuous observations gracefully: Figaro produces no Gibbs samples whereas Anglican produces very inaccurate samples. (Our preliminary testing also found Figaro an order of magnitude slower than JAGS on models with just a few discrete variables.)

6.1 Approximate inference

We report three benchmarks of approximate inference using Gibbs sampling:

(1) clustering of data points using a Gaussian mixture model (Section 3.2)
(2) supervised document classification using a Naive Bayes model [McCallum and Nigam 1998]
(3) unsupervised topic modeling using Latent Dirichlet Allocation (LDA) [Blei et al. 2003]

Gibbs sampling works by repeatedly *sweeping* through all unobserved random variables and *updating* their currently inferred values randomly. Thus a sweep consists of as many updates as there are random variables that are unobserved and uneliminated (such as unclassified data points or documents).

On each benchmark, we compare with

- AugurV2, a probabilistic-programming research system focused on composable and performant MCMC. (Applying AugurV2 required a small patch to make its algebraic rewriting more robust.)

On the first two benchmarks, we further compare with

- JAGS, a widely used probabilistic-programming system specialized for Gibbs sampling. (JAGS does not scale to the third benchmark.)

Both JAGS and AugurV2 perform different computations than Hakaru, because those systems do not eliminate latent variables [Casella and Robert 1996] as our simplification transformation does. In the first benchmark, Gaussian mixture classification, we further compare with

- STAN, another widely used probabilistic-programming system that cannot perform Gibbs sampling but defaults to a very different MCMC inference algorithm, namely HMC [Betancourt 2017; Neal 2011] with No-U-Turn Sampling [Hoffman and Gelman 2014]. (Applying STAN required the manual elimination of latent discrete array variables, a transformation automated by our simplification transformation.)

In the second benchmark, Naive Bayes document classification, we further compare with

- MALLET, a popular Java-based package for statistical natural-language processing that can be configured to perform the same computation as Hakaru.

To summarize the results across benchmarks, our generated code turns out to be faster than JAGS and MALLET, and more accurate for a given time budget than AugurV2 and STAN. As noted above, our system executes a different algorithm than JAGS, AugurV2, and STAN, which we credit for the higher eventual accuracy we achieve. We reiterate that the purpose of these benchmarks is to show that Hakaru compiles a new class of inference algorithms while maintaining competitive performance, not to rehash or analyze the superiority of a particular inference algorithm.

Gaussian mixture model. The first benchmark uses synthetic data, and we show two variations. Following the Gaussian mixture model in Section 3.2, we draw $n = 10000$ (5000) data points from a mixture of $m = 50$ (25) normal distributions, whose standard deviations are all 1 and whose means are independently generated with standard deviation $\sigma = 14$ and mean $\mu = 0$. We then hold out all the labels $\vec{y}$ and use Gibbs sampling to infer them.

We can compare inference accuracy on this benchmark, because we know the true labels of our synthetic data.[5] Figure 10 plots the accuracy achieved by each sampler against wall-clock time. Hakaru's generated code achieves higher accuracy compared to STAN's very different algorithm, and compared to JAGS and AugurV2 after a few seconds. This is the case even though, as marks

[5]For this clustering task, symmetry (unidentifiability) demands we define accuracy as the proportion of data points classified correctly under the most favorable one-to-one correspondence between true and inferred labels. Hence computing accuracy requires solving the *assignment problem*. For STAN, which samples $\vec{\theta}$ and $\vec{x}$ rather than $\vec{y}$, we plot *expected* accuracy.

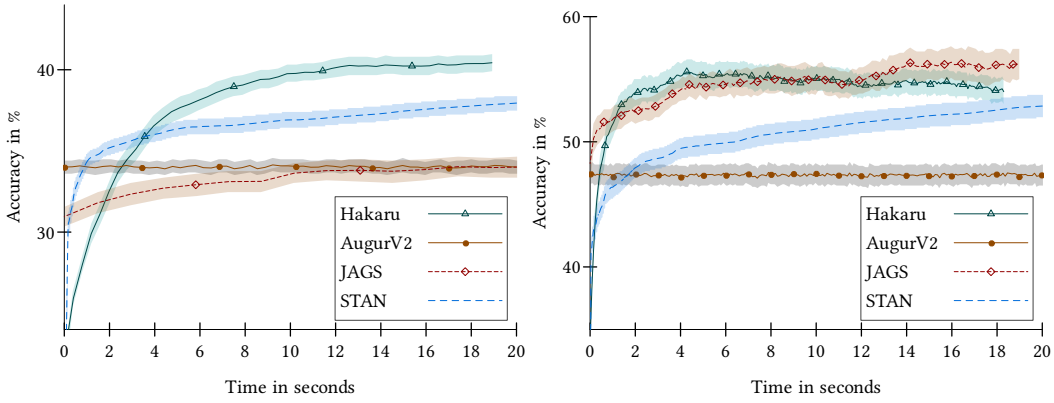

Fig. 10. Comparison of samplers for the Gaussian mixture model with $n = 10000$, $m = 50$ and with $n = 5000$, $m = 25$. Startup time is removed to Table 1. Curves represent mean accuracy over time; shaded area is standard error of 50 trials with different input data. Each mark on a curve represents 10 sweeps by Hakaru or JAGS or 100 sweeps by AugurV2. The mixture weights are drawn from the flat Dirichlet distribution, so clustering all points together would achieve accuracy ≈ 9% for $m = 50$ and ≈ 15% for $m = 25$.

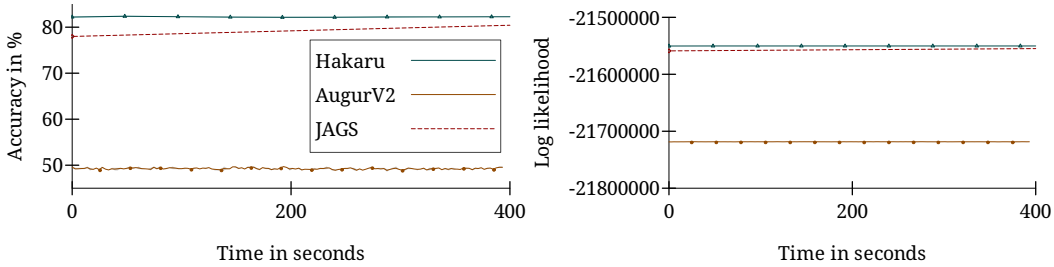

Fig. 11. Comparison of Gibbs samplers for Naive Bayes document classification. Curves represent mean accuracy or log likelihood over time; shaded area is standard error. Each mark on a curve represents 1 sweep by Hakaru or 100 sweeps by AugurV2; a sweep by JAGS takes more than 500 seconds. The documents are evenly distributed among 20 newsgroups, so a random or constant classifier would achieve 5% accuracy. Log likelihood consists of supervised and inferred labels, which are correlated due to eliminating latent variables.

on the curves show, AugurV2 is an order of magnitude faster at performing a sweep than Hakaru and JAGS. We credit our greater accuracy to simplification eliminating the latent variables $\vec{\theta}$ and $\vec{x}$ (Section 3.2). STAN works well on other models for which simplification has nothing to do.

Naive Bayes topic model. The second benchmark uses the 20 Newsgroups corpus, which consists of 19997 articles classified into 20 newsgroups [Joachims 1997]. We hold out 10% of the classifications and use Gibbs sampling to infer them, following a Dirichlet-multinomial Naive Bayes model [McCallum and Nigam 1998; Resnik and Hardisty 2010].[6]

Again we can compare inference accuracy, because we know the true labels we hold out. We also compare the log likelihood of the samples. Figure 11 plots these two metrics against wall-clock time. As the curves show, Hakaru's generated code achieves higher accuracy and likelihood right from

[6]In this model and the LDA model, to encode that different documents have different numbers of words, we use two integer arrays of equal length, one containing word identifiers and one containing document identifiers. We could as well have used a single ragged array of integer arrays, where each inner array contains the word identifiers that make up a document.

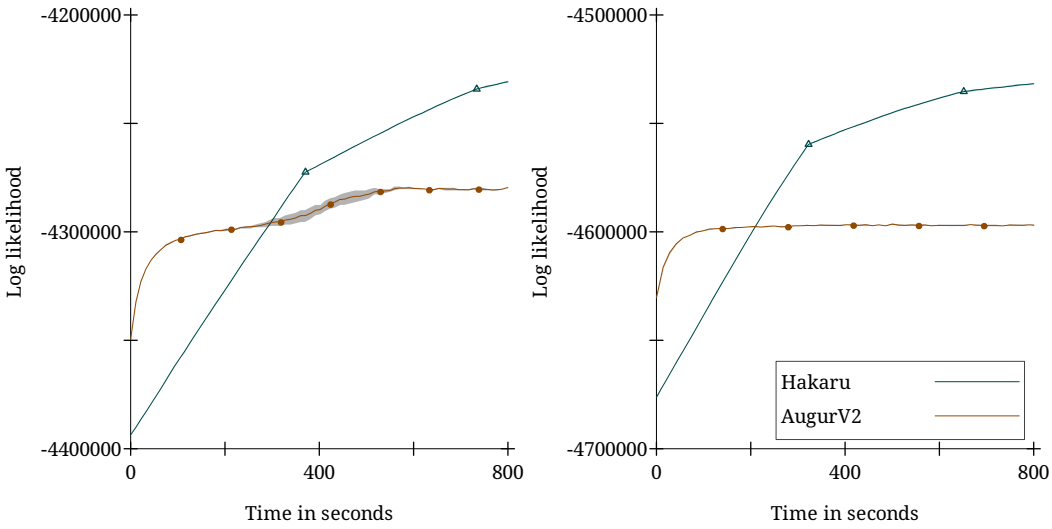

Fig. 12. Comparison of Gibbs samplers for the LDA model, with 50 and 100 topics. Curves represent mean log likelihood over time; shaded area is standard error. Each mark on a curve represents 1 sweep by Hakaru or 10 sweeps by AugurV2.

the first sweep onward. This is the case even though, as marks on the curves show, AugurV2 is two orders of magnitude faster at performing a sweep. We again credit our simplification transformation eliminating the latent variables and generating code that samples no continuous variables. That is, a sweep by our generated code is not the same mathematical operation as a sweep by AugurV2 or JAGS. However, we do not know why JAGS produces higher-quality samples than AugurV2.

For a speed comparison against inference code that has been specialized and tuned by hand for the same mathematical operation as our generated code, we also configure MALLET to compute our Gibbs updates, by calling them 19997-fold cross-validation. Our generated code is 9× as fast as MALLET, performing an update in 21.32 ± 0.04 ms while MALLET takes 189.95 ± 4.87 ms.

Latent Dirichlet Allocation topic model. The third benchmark applies the LDA model [Blei et al. 2003] to infer topics from the KOS data set [Dheeru and Karra Taniskidou 2017], which contains 467714 words drawn from a vocabulary of 6906. We do not hold out any data.

Figure 12 plots log likelihood against wall-clock time, for 50 topics and 100 topics, using Hakaru and AugurV2. Here, AugurV2 is more accurate in the first few minutes. Within 1 sweep, Hakaru's sample likelihood surpasses AugurV2's, and continues to increase past the bounds of the plot. We conclude that integrating out latent variables produces a slower but likelier result on each update.

Compilation and startup time. Time in the prior figures does not include startup: the time it takes to initialize a system or generate machine code for the given model or the given input data. Table 1 quantifies this startup time separately. On one hand, Hakaru has significant ahead-of-time compile time, because the simplification transformation can take minutes. We also incur moderate per-data startup time, for run-time specialization and machine-code generation. On the other hand, JAGS incurs negligible per-model startup time but substantial per-data startup time, because it unrolls arrays into a graph in memory before sampling. Moreover, we have observed the per-data startup time incurred by JAGS to rise faster than linearly with respect to the input data size. AugurV2, like JAGS, does not eliminate latent variables and has negligible per-model startup time, and like

Table 1. Startup time (mean and standard error in seconds) for different benchmarks and systems before sampling begins

Benchmark	System	Compile	Startup
GMM	Hakaru	545 ± 7	0.192 ± 0.002
GMM	JAGS	–	223 ± 3
GMM	AugurV2	–	0.068 ± 0.001
GMM	STAN	34.3 ± 0.1	0.641 ± 0.006
Naive Bayes	Hakaru	134 ± 6	17.61 ± 0.09
Naive Bayes	JAGS	–	22400 ± 400
Naive Bayes	AugurV2	–	0.43 ± 0.06
LDA	Hakaru	136 ± 6	2.904 ± 0.006
LDA	AugurV2	–	13.00 ± 0.08

Table 2. Run time in seconds (mean over 1000 trials and standard error) of one sweep of Gibbs sampling with $m = 50$ and $n = 10000$. Slowdown is compared to full optimization.

Optimizations	Time in seconds	Slowdown
No optimizations	471.4 ± 0.6	1848 ×
No histogram	460.6 ± 0.2	1805 ×
No LICM and loop fusion	328.7 ± 0.1	1289 ×
No loop fusion	0.471 ± 0.003	1.8×
No run-time specialization	2.422 ± 0.005	9.5×
Full optimization	0.255 ± 0.001	—

Hakaru has no size-dependent initialization. STAN incurs moderate compile and startup times, but its automatic tuning (*burn-in*) takes tens of minutes, so instead of accounting for burn-in in Table 1, we show STAN's decent performance and quick startup by plotting the beginning of burn-in in Figure 10. We were unable to improve the overall picture by reducing or disabling burn-in.

6.2 Benefits of each optimization

We perform an ablation study to show how much our optimizations benefit speed. Table 2 shows the run time of one sweep of Gibbs sampling with the larger data size used in Figure 10. We compare the time with different optimizations disabled. We disable one optimization at a time, except LICM and loop fusion because loop fusion requires LICM (Section 5.1). We never disable simplification (Section 3) because it is necessary to compile the new class of algorithms at all. Although these optimizations have a combined effect, these times give us a general idea of how individual optimizations affect overall performance.

The measurements show that our performance is made competitive by no single optimization, but rather by the conjunction of the histogram transformation and LICM: the two optimizations deliver $< 2\times$ speedup separately but $100\times$ speedup together! Also, run-time specialization and loop fusion yield $10\times$ and $2\times$ speedups respectively. We reiterate that it is in the domain of array inference algorithms that our optimizations can be aggressive and profitable.

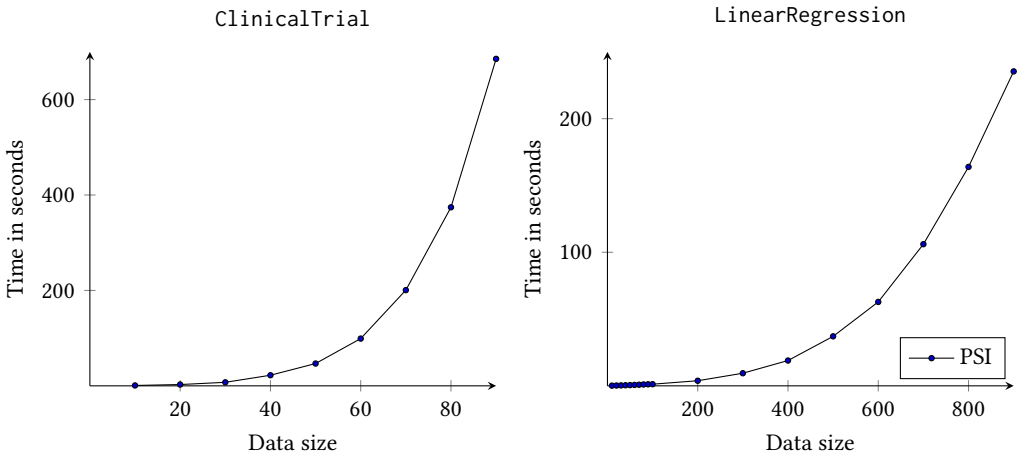

Fig. 13. PSI performance on exact-inference benchmarks, using `build-release.sh` and the `--nocheck` flag

6.3 Exact inference

To benchmark exact inference, we use the `ClinicalTrial` and `LinearRegression` examples from the R2 system [Nori et al. 2014]. The `ClinicalTrial` example infers whether a treatment is effective from the Boolean symptoms of a control group and a treated group of patients. The `LinearRegression` example fits a line to a collection of data points. In both benchmarks, Bayesian inference efficiently preserves and tracks the uncertainty of the quantities inferred. This information can be useful for making decisions under risk, and is not available through maximum-likelihood and maximum-a-posteriori estimation (such as ordinary regression).

For both benchmarks, we compare the code generated by our compilation pipeline against the code generated by the same pipeline except replacing the Sham backend (Section 5) by a previous backend that emits Haskell code. The latter code is representative of the specialized program that a practitioner would write by hand in a GPL, because array simplification (Section 3) already delivers that code as a closed-form formula in both pipelines.

- For the `ClinicalTrial` benchmark, the exact solution on 10000 data points takes 115.9 μs to compute (standard deviation 0.1 μs over 2000 trials). In contrast, the Haskell pipeline takes an average of 409.8 μs, which is 3× slower.
- For the `LinearRegression` benchmark, the exact solution on 10000 data points takes 33 μs to compute (standard deviation 4 ns over 2000 trials). In contrast, the Haskell pipeline takes an average of 363.4 μs, which is 11× slower.

These times are orders of magnitude less than even just the startup times of any approximate inference procedure.

We also compare the performance of PSI [Gehr et al. 2016], a system for exact inference that supports arrays, on the two benchmarks. Figure 13 plots PSI's run times, which increase with the data size and quickly become prohibitive, because PSI does not perform compilation and unrolls all random choices in arrays before reasoning about them. In both benchmarks, Hakaru is over 5000000× faster while handling over 10× more data. Again, the key to this efficiency is Hakaru's combination of array transformations and loop optimizations. However, also contributing to the speed difference is PSI's use of exact rational arithmetic throughout. In contrast, although Hakaru uses exact arithmetic, it generates code that uses floating-point arithmetic.

7 RELATED WORK

To situate our work in probabilistic programming, we consider which components we *specialize* using a domain-specific language and which components we *reuse* off the shelf.

The difficulty of inference is exacerbated by the ease of composing models. To address this, some systems provide a few general-purpose inference algorithms [de Salvo Braz et al. 2007; Goodman et al. 2008; Goodman and Stuhlmüller 2014; Kiselyov 2016; Lunn et al. 2000; Milch et al. 2007; Nori et al. 2014; Wingate et al. 2011; Wu et al. 2016] or restrict the language to distributions that are continuous [Carpenter et al. 2017], discrete [Kiselyov and Shan 2009; Pfeffer 2007], or relatively low-dimensional [Gehr et al. 2016]. Other systems provide a toolbox or language of inference techniques, so as to specialize inference to the given model [Fischer and Schumann 2003; Huang et al. 2017; Mansinghka et al. 2014; Pfeffer 2016; Tran et al. 2017; Tristan et al. 2014; Wood et al. 2014]. We follow the latter approach. In particular, by building on prior work on Hakaru [Narayanan et al. 2016; Zinkov and Shan 2017], we support a mix of exact and approximate inference by reusing program transformations such as simplification and disintegration on model and inference alike.

Many sophisticated probabilistic programming systems end up (re)implementing computer algebra [de Salvo Braz and O'Reilly 2017; de Salvo Braz et al. 2016; Fischer and Schumann 2003; Gehr et al. 2016; Huang et al. 2017; Tristan et al. 2014]. Reusing an existing computer algebra system and specializing it to the language of patently linear expressions makes it possible to eliminate latent variables and recognize primitive distributions without hard-coding patterns such as conjugacy relationships [Carette and Shan 2016]. We extend the latter approach to arrays, further reusing computer algebra to solve equations in our key unproduct operation. Our histogram optimization seems related to transforming loops into list homomorphisms (map-reduce), but we could not find or reuse any work that makes this relationship clear.

Most probabilistic programming systems either interpret their programs, or compile or embed them through a GPL. Generating GPU code has also been shown beneficial [Huang et al. 2017; Tristan et al. 2014]. In contrast, we generate optimized code through LLVM, but specialize our code generation to take advantage of pure array programs and map-reduce loops.

ACKNOWLEDGMENTS

We thank Allen Riddell for help with STAN. We also thank our anonymous reviewers at PLDI 2018, ICFP 2018, POPL 2019, PLDI 2019, *and* ICFP 2019.

This research was supported by DARPA contract FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc. (through its support for the Indiana University Pervasive Technology Institute), and the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Thomas Bayes. 1763. An Essay towards Solving a Problem in the Doctrine of Chances. *Philosophical Transactions of the Royal Society of London* 53 (1763), 370–418.

Michael Betancourt. 2017. *A Conceptual Introduction to Hamiltonian Monte Carlo.* e-Print 1701.02434. arXiv.org. https://arxiv.org/abs/1701.02434

David Blackwell. 1947. Conditional Expectation and Unbiased Sequential Estimation. *The Annals of Mathematical Statistics* 18, 1 (March 1947), 105–110.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* 3, Jan. (Jan. 2003), 993–1022.

Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio V. Russo, Adam Scibior, and Marcin Szymczak. 2016. Fabular: Regression Formulas as Probabilistic Programming. In *Proceedings of the 43th Symposium on Principles of Programming*

Languages (POPL). ACM Press, 271–283.

Wray L. Buntine. 1994. Operations for Learning with Graphical Models. *Journal of Artificial Intelligence Research* 2 (1994), 159–225.

Jacques Carette and Chung-chieh Shan. 2016. Simplifying Probabilistic Programs Using Computer Algebra. In *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016 (Lecture Notes in Computer Science)*, Marco Gavanelli and John H. Reppy (Eds.). 135–152.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32.

George Casella and Christian P. Robert. 1996. Rao-Blackwellisation of Sampling Schemes. *Biometrika* 83, 1 (1996), 81–94.

Frédéric Chyzak and Bruno Salvy. 1998. Non-commutative Elimination in Ore Algebras Proves Multivariate Holonomic Identities. *Journal of Symbolic Computation* 26, 2 (1998), 187–227.

Samantha R. Cook, Andrew Gelman, and Donald B. Rubin. 2006. Validation of Software for Bayesian Models Using Posterior Quantiles. *Journal of Computational and Graphical Statistics* 15, 3 (2006), 675–692.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, Manuela M. Veloso (Ed.). 2462–2467.

Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. 2007. Lifted First-Order Probabilistic Inference. In *Introduction to Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press, 433–451.

Rodrigo de Salvo Braz and Ciaran O'Reilly. 2017. Exact Inference for Relational Graphical Models with Interpreted Functions: Lifted Probabilistic Inference Modulo Theories, Gal Elidan, Kristian Kersting, and Alexander T. Ihler (Eds.). AUAI Press.

Rodrigo de Salvo Braz, Ciaran O'Reilly, Vibhav Gogate, and Rina Dechter. 2016. Probabilistic Inference Modulo Theories. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, Subbarao Kambhampati (Ed.). AAAI Press, 3591–3599. http://www.ijcai.org/Abstract/16/506

Rina Dechter. 1998. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In *Learning and Inference in Graphical Models*, Michael I. Jordan (Ed.). Kluwer, Dordrecht. Paperback: *Learning in Graphical Models*, MIT Press.

Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

Bernd Fischer and Johann Schumann. 2003. AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. *Journal of Functional Programming* 13, 3 (2003), 483–508.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Proceedings of the 28th International Conference on Computer Aided Verification, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83.

Alan E. Gelfand and Adrian F. M. Smith. 1990. Sampling-Based Approaches to Calculating Marginal Densities. *J. Amer. Statist. Assoc.* 85, 410 (1990), 398–409.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2014. *Bayesian Data Analysis* (third ed.). CRC Press.

John Geweke. 2004. Getting It Right. *J. Amer. Statist. Assoc.* 99, 467 (2004), 799–804.

Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*, Bernhard Banaschewski (Ed.). Springer, 68–85.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, David Allen McAllester and Petri Myllymäki (Eds.). 220–229.

Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

Thomas L. Griffiths and Mark Steyvers. 2004. Finding Scientific Topics. *Proceedings of the National Academy of Sciences* 101, suppl 1 (2004), 5228–5235. https://www.pnas.org/content/101/suppl_1/5228

Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.

Matthew D. Hoffman, Matthew J. Johnson, and Dustin Tran. 2018. Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. In *Advances in Neural Information Processing Systems*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10739–10749. http://papers.nips.cc/paper/8270-autoconj-recognizing-and-exploiting-conjugacy-without-a-domain-specific-language.pdf

Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *PLDI '17: Proceedings of the ACM Conference on Programming Language Design and Implementation*, Albert Cohen and Martin T. Vechev (Eds.). ACM Press, 111–125.

Thorsten Joachims. 1997. A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 143–151. http://dl.acm.org/citation.cfm?id=645526.657278

Manuel Kauers. 2013. The Holonomic Toolkit. In *Computer Algebra in Quantum Field Theory*, Carsten Schneider and Johannes Blümlein (Eds.). Springer, 119–144.

Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In *Proceedings of APLAS 2016: 14th Asian Symposium on Programming Languages and Systems (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.). Springer, 357–376.

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded Probabilistic Programming. In *Proceedings of the Working Conference on Domain-Specific Languages (Lecture Notes in Computer Science)*, Walid Mohamed Taha (Ed.). Springer, 360–384.

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Andrey N. Kolmogorov. 1950. Unbiased Estimates. *Izvestiya Akademii Nauk SSSR Seriya Matematicheskaya* 14, 4 (1950), 303–326.

Jun S. Liu. 1994. The Collapsed Gibbs Sampler in Bayesian Computations with Applications to a Gene Regulation Problem. *J. Amer. Statist. Assoc.* 89, 427 (1994), 958–966.

Jun S. Liu, Wing Hung Wong, and Augustine Kong. 1994. Covariance Structure of the Gibbs Sampler with Applications to the Comparisons of Estimators and Augmentation Schemes. *Biometrika* 81, 1 (1994), 27–40.

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. WinBUGS—A Bayesian Modelling Framework: Concepts, Structure, and Extensibility. *Statistics and Computing* 10, 4 (2000), 325–337.

David J. C. MacKay. 1998. Introduction to Monte Carlo Methods. In *Learning and Inference in Graphical Models*, Michael I. Jordan (Ed.). Kluwer, Dordrecht. Paperback: *Learning in Graphical Models*, MIT Press.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. *Venture: a Higher-Order Probabilistic Programming Platform with Programmable Inference*. e-Print 1404.0099. arXiv.org.

Andrew McCallum and Kamal Nigam. 1998. A Comparison of Event Models for Naive Bayes Text Classification. In *AAAI-98 workshop on learning for text categorization*, Vol. 752. 41–48.

Andrew Kachites McCallum. 2002. MALLET: A Machine Learning for Language Toolkit. http://mallet.cs.umass.edu

Xiao-Li Meng and David A. van Dyk. 1999. Seeking Efficient Data Augmentation Schemes via Conditional and Marginal Augmentation. *Biometrika* 86, 2 (1999), 301–320.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic Models with Unknown Objects. In *Introduction to Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press, Chapter 13, 373–398.

Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Proceedings of AISTATS 2018: 21st International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Amos Storkey and Fernando Perez-Cruz (Eds.). 1037–1046.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Proceedings of FLOPS 2016: 13th International Symposium on Functional and Logic Programming (Lecture Notes in Computer Science)*, Oleg Kiselyov and Andy King (Eds.). Springer, 62–79.

Praveen Narayanan and Chung-chieh Shan. 2017. Symbolic Conditioning of Arrays in Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 11:1–11:25.

Radford M. Neal. 2011. MCMC Using Hamiltonian Dynamics. In *Handbook of Markov Chain Monte Carlo*, Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng (Eds.). CRC Press, Chapter 5.

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, Carla E. Brodley and Peter Stone (Eds.). AAAI Press, 2476–2482.

Fritz H. Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, and Noah Goodman. 2018. Automated Enumeration of Discrete Latent Variables. (2018). Poster at PROBPROG 2018.

Anand Patil, David Huard, and Christopher J. Fonnesbeck. 2010. PyMC: Bayesian Stochastic Modelling in Python. *Journal of Statistical Software* 35, 4 (July 2010), 1–81.

Karl Pearson. 1894. III. Contributions to the Mathematical Theory of Evolution. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 185 (1894), 71–110. https://doi.org/10.1098/rsta.1894.0003 arXiv:http://rsta.royalsocietypublishing.org/content/185/71.full.pdf

Avi Pfeffer. 2007. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. In *Introduction to Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press, Chapter 14, 399–432.

Avi Pfeffer. 2016. *Practical Probabilistic Programming.* Manning Publications.

Martyn Plummer. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing.*

David Pollard. 2001. *A User's Guide to Measure Theoretic Probability.* Cambridge University Press.

David Poole and Nevin Lianwen Zhang. 2003. Exploiting Contextual Independence In Probabilistic Inference. *Journal of Artificial Intelligence Research* 18 (2003), 263–313.

Lawrence R. Rabiner. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE* 77, 2 (Feb. 1989), 257–286.

Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL).* ACM Press, 154–165.

C. Radhakrishna Rao. 1945. Information and Accuracy Attainable in the Estimation of Statistical Parameters. *Bulletin of the Calcutta Mathematical Society* 37, 3 (1945), 81–91.

Philip Resnik and Eric Hardisty. 2010. *Gibbs Sampling for the Uninitiated.* Technical Report CS-TR-4956 UMIACS-TR-2010-04 LAMP-TR-153. University of Maryland.

Scott Sanner and Ehsan Abbasnejad. 2012. Symbolic Variable Elimination for Discrete and Continuous Graphical Models. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, Jörg Hoffmann and Bart Selman (Eds.). AAAI Press, 1954–1960.

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Proceedings of the 44th Symposium on Principles of Programming Languages (POPL).* ACM Press, 130–144.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems: Proceedings of ESOP 2017, 26th European Symposium on Programming (Lecture Notes in Computer Science)*, Yang Hongseok (Ed.). Springer, 855–879.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. *Deep Probabilistic Programming.* e-Print 1701.03757. arXiv.org. 5th International Conference on Learning Representations.

Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen J. Green, and Guy Lewis Steele, Jr. 2014. *Augur: a Modeling Language for Data-Parallel Probabilistic Inference.* e-Print 1312.3613. arXiv.org. http://arxiv.org/abs/1312.3613

Deepak Venugopal and Vibhav Gogate. 2013. Dynamic Blocking and Collapsing for Gibbs Sampling. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence*, Ann Nicholson and Padhraic Smyth (Eds.). 664–673.

Herbert S. Wilf and Doron Zeilberger. 1992. An Algorithmic Proof Theory for Hypergeometric (Ordinary and "q") Multi-sum/Integral Identities. *Inventiones mathematicae* 108 (1992), 557–633.

David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of AISTATS 2011: 14th International Conference on Artificial Intelligence and Statistics (JMLR Workshop and Conference Proceedings)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). MIT Press, 770–778.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of AISTATS 2014: 17th International Conference on Artificial Intelligence and Statistics (JMLR Workshop and Conference Proceedings).* 1024–1032.

Yi Wu, Lei Li, Stuart J. Russell, and Rastislav Bodík. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, Subbarao Kambhampati (Ed.). AAAI Press, 3637–3645. http://www.ijcai.org/Abstract/16/512

Nevin Lianwen Zhang and David L. Poole. 1994. A Simple Approach to Bayesian Network Computations. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence.* 171–178.

Nevin Lianwen Zhang and David L. Poole. 1996. Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research* 5 (1996), 301–328.

Robert Zinkov and Chung-chieh Shan. 2017. Composing Inference Algorithms as Program Transformations, Gal Elidan, Kristian Kersting, and Alexander T. Ihler (Eds.). AUAI Press.