Symbolic Conditioning of Arrays in Probabilistic Programs

PRAVEEN NARAYANAN, Indiana University, USA
CHUNG-CHIEH SHAN, Indiana University, USA

Probabilistic programming systems make machine learning more modular by automating *inference*. Recent work by Shan and Ramsey makes inference more modular by automating *conditioning*. Their technique uses a symbolic program transformation that treats conditioning generally via the measure-theoretic notion of *disintegration*. This technique, however, is limited to conditioning a single scalar variable. As a step towards modular inference for realistic machine learning applications, we have extended the disintegration algorithm to symbolically condition arrays in probabilistic programs. The extended algorithm implements *lifted disintegration*, where repetition is treated symbolically and without unrolling loops. The technique uses a language of *index variables* for tracking expressions at various array levels. We find that the method works well for arbitrarily-sized arrays of independent random choices, with the conditioning step taking time linear in the number of indices needed to select an element.

CCS Concepts: • **Mathematics of computing** → **Bayesian computation**; • **Theory of computation** → **Functional constructs**; • **Computing methodologies** → *Symbolic calculus algorithms*;

Additional Key Words and Phrases: probabilistic programs, conditional measures, arrays

ACM Reference Format:
Praveen Narayanan and Chung-chieh Shan. 2017. Symbolic Conditioning of Arrays in Probabilistic Programs. *Proc. ACM Program. Lang.* 1, ICFP, Article 11 (September 2017), 25 pages.
https://doi.org/10.1145/3110255

1 INTRODUCTION

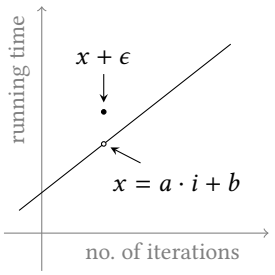
The guiding principle of machine learning is that of improving from experience. By using data to enhance our knowledge of the world, we can perform more effectively at tasks such as predicting the topics of documents or calculating the chance of heart attacks. The task at hand often determines the formalism for mechanically learning from data, and *Bayesian* techniques in particular are useful across a large class of problems.

Bayesian analyses consist of *modeling* followed by *inference*. In the modeling step we use probability distributions to express our beliefs prior to observing any data. In the inference step we answer questions posterior to observing data.

For example, say we want to analyze the running time of an iterative algorithm. We might model the algorithm as taking some initialization time along with some constant time per iteration. A *Bayesian linear regression* model would suit this analysis.

Fig. 1 shows a Bayesian linear regression model. The goal of linear regression is to fit a line—characterized by its slope a and intercept b—through a collection of points. In the Bayesian setting, we develop this by:

2475-1421/2017/9-ART11
https://doi.org/10.1145/3110255



$$a \sim \mathcal{N}(\mu_a,\ \sigma_a)$$
$$b \sim \mathcal{N}(\mu_b,\ \sigma_b)$$
$$x = a \cdot i + b$$
$$\epsilon \sim \mathcal{N}(0,\ 1)$$

Fig. 1. A canonical sample (left) of a regression model (right) for some value of a, b, i, and ϵ. We observe $x + \epsilon$ and want to infer a distribution over a and b using this model.

- stating our prior beliefs about the slope and intercept—modeled here as Gaussian distributions with fixed means (μ_a, μ_b) and standard deviations (σ_a, σ_b)—and
- acknowledging the noise inherent in our measurements—modeled by ϵ here as a standard Gaussian distribution.

The variables a and b map to the time-per-iteration and the initialization time respectively. The variable i maps to the number of iterations for which we run our algorithm. Ideally we would run this experiment for various values of i—and this motivates our paper—but it is nevertheless instructive to understand how a Bayesian analysis is informed by a single data-point.

Having specified our model, we move on to the inference step, where we want to answer questions about the line $(a \cdot i + b)$ in light of a noisy measurement of the running time $(x + \epsilon)$. Inference techniques require three quantities – the model, a data-introducing condition such as $x + \epsilon = t$, and the question of interest such as "What is the expected value of a?", or "What is the most likely value of b?". Inference consists of *conditioning*, which combines the model and the condition, followed by *querying*, which processes the question of interest.

Although modeling and inference are separate steps in theory, implementations of Bayesian techniques intertwine the two in practice. To classify documents by their topics, for example, the prevalent approach is to customize an inference algorithm towards a particular model [Asuncion et al. 2009; Resnik and Hardisty 2009]. Consequently, a small change to the task on paper (adjusting the model) can lead to a disproportionately large change to the implementation in code.

Probabilistic programming is about decoupling models and inference. With the advent of probabilistic programming languages, we are able to reuse an inference method on different models [Goodman et al. 2008; Milch et al. 2007], and try out different inference methods on a fixed model [Mansinghka et al. 2014; Narayanan et al. 2016; Wood et al. 2014]. This separation frees the implementor to explore trade-offs between intricate models and sophisticated inference methods.

Similarly, although conditioning and querying are separate steps of inference in theory, implementations of inference intertwine the two in practice. Shan and Ramsey [2017] show that we can decouple conditioning and querying using the measure-theoretic notion of *disintegration* (as advocated by Chang and Pollard [1997]). We adopt this approach and implement a modular inference technique where the first part of inference performs conditioning and returns a distribution that will be separately queried by the second part of inference.

Currently, modular inference techniques that separate out conditioning can only be used for small quantities of observational data. The existing methods simply unroll larger structures [Gehr et al. 2016; Shan and Ramsey 2017], resulting in severe code growth. We desire an automatic conditioning method that observes multiply-indexed arrays of data symbolically and without unrolling. In this paper we achieve this goal via a technique that extends disintegration to handle arrays.

Before we describe array disintegration (in sections 4 and 5), we set up disintegration on a language without arrays (in sections 2 and 3). We use the running example in the rest of this section to describe modeling and inference in the context of probabilistic programming languages.

1.1 Models as Probabilistic Programs

Probabilistic programs express Bayesian models. Let us revisit the Bayesian linear regression model, which we can express as the following probabilistic program:

$$
\begin{aligned}
&\textbf{blr} : \ \mathbb{M}\ (\mathbb{R} \times (\mathbb{R} \times \mathbb{R})) \\
&\textbf{blr} = \textbf{do}\ \{a \leftsquigarrow \textbf{normal}\ \mu_a\ \sigma_a; \\
&\qquad\qquad b \leftsquigarrow \textbf{normal}\ \mu_b\ \sigma_b; \\
&\qquad\qquad x \leftsquigarrow \textbf{return}\ (a \cdot i + b); \\
&\qquad\qquad \epsilon \leftsquigarrow \textbf{normal}\ 0\ 1; \\
&\qquad\qquad \textbf{return}\ (x + \epsilon\,,\,(a\,,\,b))\}
\end{aligned}
\tag{1}
$$

This program is written in a probabilistic programming language that we call core Hakaru. Fig. 2 defines its syntax.

$$
\begin{aligned}
&\text{Real numbers } r \in \mathbb{R} \qquad \text{Variables } x \\
&\text{Terms } e, m, M ::= x \mid r \mid -e \mid e + e \mid e \cdot e \mid \textbf{dGauss}\ e\ e\ e \mid () \mid (e\,,\,e) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\
&\qquad\qquad \mid \textbf{lebesgue} \mid \textbf{normal}\ e\ e \mid \textbf{return}\ e \mid \textbf{do}\ \{x \leftsquigarrow m; M\} \mid \textbf{do}\ \{\textbf{factor}\ e; M\} \\
&\text{Types} \qquad \alpha, \beta ::= \mathbb{1} \mid \mathbb{R} \mid \alpha \times \beta \mid \mathbb{M}\ \alpha
\end{aligned}
$$

Fig. 2. Syntactic forms of core Hakaru

The program in eq. (1) is composed of nested **do** $\{x \leftsquigarrow e; e\}$ constructs, for which we use this shorthand notation:

$$
\textbf{do}\ \{x_1 \leftsquigarrow m_1; \textbf{do}\ \{x_2 \leftsquigarrow m_2; \ldots; \textbf{do}\ \{x_n \leftsquigarrow m_n; M\} \ldots\}\} \equiv \textbf{do}\ \{x_1 \leftsquigarrow m_1; x_2 \leftsquigarrow m_2; \ldots; x_n \leftsquigarrow m_n; M\}. \tag{2}
$$

Core Hakaru programs are written in a *mochastic* (monadic-stochastic) style. Just as a monadic *bind* operator assigns a variable to the result of an effectful computation, the core Hakaru "$\leftsquigarrow$" operator defines a variable distributed according to the expression on the right hand side. As a special case, the combination of $\leftsquigarrow$ and the **return** primitive corresponds to *let-binding* (written as = in the original model).

Terms that denote measures (which generalize distributions) have type $\mathbb{M}$ in core Hakaru. Fig. 3 describes the typing rules for terms of measure type; the omitted rules are standard.

$$
\frac{}{\Gamma \vdash \textbf{lebesgue} : \mathbb{M}\ \mathbb{R}} \qquad \frac{\Gamma \vdash e : \mathbb{R} \quad \Gamma \vdash e' : \mathbb{R}}{\Gamma \vdash \textbf{normal}\ e\ e' : \mathbb{M}\ \mathbb{R}} \qquad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \textbf{return}\ e : \mathbb{M}\ \alpha}
$$

$$
\frac{\Gamma \vdash m : \mathbb{M}\ \alpha \quad \Gamma, x{:}\alpha \vdash M : \mathbb{M}\ \beta}{\Gamma \vdash \textbf{do}\ \{x \leftsquigarrow m; M\} : \mathbb{M}\ \beta} \qquad \frac{\Gamma \vdash e : \mathbb{R} \quad \Gamma \vdash M : \mathbb{M}\ \alpha}{\Gamma \vdash \textbf{do}\ \{\textbf{factor}\ e; M\} : \mathbb{M}\ \alpha}
$$

Fig. 3. Typing rules for terms of measure type

The final line of the **blr** program in eq. (1) returns the noisy reading of the run-time paired with the corresponding slope and intercept. This agrees with the type $\mathbb{M}\ (\mathbb{R} \times (\mathbb{R} \times \mathbb{R}))$, which states that the program denotes a measure over nested tuples of real numbers.

We can say that the program in eq. (1) represents the *joint measure* $P(x + \epsilon, a, b)$. The joint decomposes into the *prior* $P(a, b)$, and the *likelihood* $P(x + \epsilon \mid a, b)$:

$$P(x + \epsilon, a, b) = P(a, b) \otimes P(x + \epsilon \mid a, b). \tag{3}$$

Let us define each element of the right-hand-side of eq. (3).

The prior $P(a, b)$ represents what we believe, before observing data, about how a and b are distributed. In **blr** we express this as a program of type $\mathbb{M}\ (\mathbb{R} \times \mathbb{R})$:

$$P(a, b) = \textbf{do}\ \{a \leftsquigarrow \textbf{normal}\ \mu_a\ \sigma_a;\ b \leftsquigarrow \textbf{normal}\ \mu_b\ \sigma_b;\ \textbf{return}\ (a, b)\}. \tag{4}$$

The likelihood $P(x + \epsilon \mid a, b)$ is how we believe a measurement of the running time depends on a and b. It is a *conditional measure*, and we can express it as function from (a, b) to a program of type $\mathbb{M}\ \mathbb{R}$:

$$P(x + \epsilon \mid a, b) = \lambda(a, b).\ \textbf{do}\ \{x \leftsquigarrow \textbf{return}\ (a \cdot i + b);\ \epsilon \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{return}\ (x + \epsilon)\}. \tag{5}$$

To create the joint measure, the factors in eq. (3) must be linked together by an operation on measures that is notated $\otimes$ and pronounced "bind x":

$$\otimes : \mathbb{M}\ \alpha \to (\alpha \to \mathbb{M}\ \beta) \to \mathbb{M}\ (\beta \times \alpha)$$
$$m \otimes f = \textbf{do}\ \{a \leftsquigarrow m;\ b \leftsquigarrow f\ a;\ \textbf{return}\ (b, a)\}.$$

As this definition illustrates, monadic bind is an important construct for composing larger programs out of smaller building blocks in general purpose languages, and core Hakaru applies the same principle in the probabilistic setting.

The grammar in fig. 2 contains some constructs that we have not yet discussed. There are routine operators for arithmetic (–) and destructors for pair types (**fst**, **snd**). The **lebesgue** construct defines the Lebesgue measure—the unique-up-to-scale translation invariant measure on $\mathbb{R}$.

To understand **factor**, we note that measures can often be re-written in terms of **lebesgue** and **factor**. Formally, this means that for some measures m, there is a function $f : \mathbb{R} \to \mathbb{R}^+$ such that:

$$m \equiv \textbf{do}\ \{x \leftsquigarrow \textbf{lebesgue};\ \textbf{factor}\ (f\ x);\ \textbf{return}\ x\}. \tag{6}$$

This is what it means for m to have a *density* f *with respect to the Lebesgue measure.*

We can use **factor** in conjunction with **lebesgue** to express a large class of measures over the reals. For example, core Hakaru provides the **dGauss** construct: **dGauss** $\mu\ \sigma\ x$ denotes the density of $\mathcal{N}(\mu, \sigma)$ at x with respect to **lebesgue**. This gives us an alternate definition for the Gaussian distribution:

$$\textbf{normal} : \mathbb{R} \to \mathbb{R} \to \mathbb{M}\ \mathbb{R} \tag{7}$$
$$\textbf{normal}\ \mu\ \sigma \equiv \textbf{do}\ \{x \leftsquigarrow \textbf{lebesgue};\ \textbf{factor}\ (\textbf{dGauss}\ \mu\ \sigma\ x);\ \textbf{return}\ x\}.$$

For more intuition about **factor** we can interpret a core Hakaru program as an importance sampler for its underlying measure. Now each sample from an importance sampler is paired with a non-negative weight. This weight is 1 for the primitive measures provided by the language, and is multiplied by the argument to **factor** in programs that use the construct.

When run as a sampler, the program in eq. (7) produces real-valued samples, but how do we know that they follow the correct Gaussian distribution? The **factor** construct assures this by re-weighting the sample by the density of $\mathcal{N}(\mu, \sigma)$ at that value. While each real value is as likely to show up as any other (as they are all being roughly drawn from **lebesgue**), the importance weights will reshape the distribution of samples to approximate the required Gaussian distribution.

We note that sampling from programs that use **lebesgue** and **factor** is a good approach to intuition but a bad one for inference. This is largely because of the impossibility of sampling from the canonical distribution on reals associated with the Lebesgue measure. We can handle this problem separately with a transformation that recognizes densities and converts a program that

uses **lebesgue** to one that uses probability measures having well-known sampling algorithms. The Hakaru probabilistic programming system provides such a simplification transformation [Carette and Shan 2016], typically used in the inference step to obtain more efficient samplers.

1.2 Inference on Probabilistic Programs

The modeling constructs shown thus far are largely consistent across probabilistic programming systems; the differences lie in the inference algorithms.

Typically we observe data for a subset of the variables in our model. While the **blr** model describes a measure over possible values of the variables a, b, and $x + \epsilon$, we observe data only for the variable $x + \epsilon$. The observed data restricts the space of possibilities, and we are interested in how a and b behave in this restricted space. In Bayesian terminology, we are interested in the *posterior distribution*.

Inference is about computing queries on the posterior. Now, many queries on a distribution can be approximately answered given a set of samples from that distribution. Thus we can intuitively perform inference by generating samples according to the model, throwing away those that are incompatible with our observations (by assigning to such samples a weight of 0), and computing our queries on the resultant set. This technique is also known as *rejection sampling*.

Although rejection sampling is useful for pedagogy, the inefficiency of wasted work makes it not so useful in practice. Additionally, the method cannot be used for observing continuous variables, when there is zero probability of matching our observations exactly.

In practice, probabilistic programming systems implement more efficient inference techniques to obtain and query posterior distributions. Systems have adapted approximate methods based on Markov Chain Monte Carlo (MCMC) [Gelman et al. 2015; Milch et al. 2007; Wingate et al. 2011] and variational inference [Kucukelbir et al. 2015; Wingate and Weber 2013], as well as exact techniques based on abstract interpretation, factor graphs, data flow analysis, and Bayes nets [Claret et al. 2013; McCallum et al. 2009; Minka et al. 2014].

Although there are various inference techniques, across them lies the same conceptual step of first obtaining the posterior distribution. This step is called *conditioning*. Every inference method first conditions the joint to obtain the posterior, and then converts the posterior into a unique representation suitable for querying.

Thus, a modular approach to inference would decouple the conditioning and the querying steps. Most systems however do not harness this modularity. Inference methods typically take three inputs—model, data, and query of interest—and repeat the work needed to perform conditioning. For example, consider a function **mh** that implements *Metropolis-Hastings* MCMC sampling (described in its most general form by Tierney [1998]) and has the following type:

$$\mathbf{mh} : \mathbb{M}\ (\alpha \times \beta) \to \alpha \to ([\beta] \to \gamma) \to \gamma. \tag{8}$$

This function takes a joint measure (the model) as its first argument and constrains its first dimension with an observation (the data) given by the second argument. The third argument is a function used for querying a set of samples (represented as $[\beta]$) and producing some output γ. Implicitly, this inference method performs conditioning, obtains the posterior *as a set of samples*, and then queries that set.

Suppose that in our running example we observe $x + \epsilon = t$, and that we want to compute the *expected value* of a given this observation. Given a function **expect** of type $[\mathbb{R} \times \mathbb{R}] \to \mathbb{R}$, we could use Metropolis-Hastings sampling to approximate this value:

$$a_{expected} = \mathbf{mh}\ \mathbf{blr}\ t\ \mathbf{expect}. \tag{9}$$

Inference based on a slightly different method like *Gibbs sampling* [explained by Casella and George 1992; Resnik and Hardisty 2009] would have a type very similar to the one for **mh**. A **gibbs** method could even share the code used by **mh** to implement conditioning—after all both methods query a set of samples. Inference based on an exact method such as abstract interpretation, however, would need a different representation of the posterior, and might end up significantly duplicating the work done to perform conditioning.

We thus seek a reusable conditioning tool that produces the posterior in a format that can be manipulated by different inference methods. We choose this format to be a program in core Hakaru, and implement a conditioning program transformation that when given data takes us from a core Hakaru program representing the model to one representing the posterior. Let us specify this transformation formally and look at examples of what it would produce.

2 CONDITIONING VIA DISINTEGRATION

Let us specify a conditioning transformation on core Hakaru programs. To perform conditioning on our running example, we seek a function that when given the joint $P(x+\epsilon, a, b)$, produces the posterior $P(a, b \mid x+\epsilon)$. The posterior is a conditional measure, i.e., a function from the observed running time $x+\epsilon$ to a measure over possible values of (a, b).

A commonly accepted specification for conditioning comes from the measure-theoretic concept of *disintegration* [covered in Billingsley 1995; Pollard 2001]:

$$P(x+\epsilon) \otimes P(a, b \mid x+\epsilon) = P(x+\epsilon, a, b). \tag{10}$$

When this relation holds, we say that the *marginal* measure $P(x+\epsilon)$ and the *posterior* measure $P(a, b \mid x+\epsilon)$ form a disintegration of the joint measure $P(x+\epsilon, a, b)$. We could motivate this specification by appealing to intuition from elementary probability theory. Recall that if the terms $P(x+\epsilon)$, $P(a, b \mid x+\epsilon)$, and $P(x+\epsilon, a, b)$ denoted *probabilities* rather than measures, Bayes' rule is precisely eq. (10) with $\otimes$ replaced by multiplication $(\cdot)$.

Previously, we saw that the terms in eq. (3) symbolized the core Hakaru programs in eqs. (4) and (5). Here too, the terms in eq. (10) symbolize core Hakaru programs; this time we want to automatically derive them. We use these symbols in a specification, expressed in core Hakaru by expanding the definition of $\otimes$:

$$\textbf{do } \{t \leftsquigarrow P(x+\epsilon); p \leftsquigarrow P(a, b \mid x+\epsilon)\ t; \textbf{ return } (t\,,p)\} \equiv P(x+\epsilon, a, b). \tag{11}$$

This specifies a conditioning transformation for the **blr** example. We can generalize this to specify a conditioning transformation for all core Hakaru programs:

$$\textbf{do } \{t \leftsquigarrow \mu; p \leftsquigarrow \kappa\ t; \textbf{ return } (t\,,p)\} \equiv m. \tag{12}$$

If we build a tool that, when given a core Hakaru program m produces the program μ and function κ such that eq. (12) is satisfied, we will know that the tool implements conditioning.

Now, if we are able to find a marginal and a posterior measure such that eq. (12) (and thus eq. (11)) is satisfied, and if we can assume that the marginal has a density f with respect to **lebesgue**, then we are better off building a transformation—let us call it **disintegrate**—that satisfies a slightly different specification:

$$\textbf{do } \{t \leftsquigarrow \textbf{lebesgue}; p \leftsquigarrow \textbf{disintegrate } m\ t; \textbf{ return } (t\,,p)\} \equiv m \tag{13}$$

Eq. (13) falls out from our assumptions:

$$
\begin{aligned}
& m \\
& \equiv \text{[eq. (12)]} \\
& \textbf{do } \{t \leftsquigarrow \mu;\ p \leftsquigarrow \kappa\ t;\ \textbf{return } (t\,,p)\} && (14) \\
& \equiv \text{[definition of density, eq. (6)]} \\
& \textbf{do } \{t \leftsquigarrow \textbf{do } \{t \leftsquigarrow \textbf{lebesgue};\ \textbf{factor } (f\ t);\ \textbf{return } t\};\ p \leftsquigarrow \kappa\ t;\ \textbf{return } (t\,,p)\} && (15) \\
& \equiv \text{[monad laws]} \\
& \textbf{do } \{t \leftsquigarrow \textbf{lebesgue};\ p \leftsquigarrow \textbf{do } \{\textbf{factor } (f\ t);\ \kappa\ t\};\ \textbf{return } (t\,,p)\}. && (16)
\end{aligned}
$$

Our assumption that the marginal has a density with respect to **lebesgue** is not unrealistic—most marginals over $\mathbb{R}$ that arise naturally in probabilistic programs have this property. It is certainly true of $P(x + \epsilon)$ in the running example.

Additionally, in eq. (16), the right hand side of the expression $p \leftsquigarrow \ldots$ is not the true posterior, but rather something *proportional* to it. Fortunately, this is adequate for a large class of querying algorithms [Tierney 1998]. We thus require that **disintegrate** produce this term generally for all input measures m, giving us the specification in eq. (13).

The specification in eq. (13) hints at the type for disintegration with respect to **lebesgue**:

$$\textbf{disintegrate} : \mathbb{M}\ (\mathbb{R} \times \beta) \to \mathbb{R} \to \mathbb{M}\ \beta. \tag{17}$$

Disintegrations do not always exist, and when they do they are not unique. Ackerman et al. [2011] show that a disintegrator must fail to find solutions for a language that can express all and only *computable* distributions. Core Hakaru is not expressive enough to represent all computable distributions, so it is unclear what disintegrations can be expressed. In our system we express failure and non-uniqueness of disintegration by having the type of **disintegrate** return a *list* of possible answers. For ease of exposition we omit this detail in the paper.

Disintegrating the Regression Example. For the **blr** example, our implemented **disintegrate** produces the following program:

$$
\begin{aligned}
\textbf{disintegrate blr} \equiv \lambda t.\ \textbf{do } \{ & a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a; \\
& b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b; \\
& \textbf{factor } (\textbf{dGauss } 0\ 1\ (t - (a \cdot i + b))); \\
& \textbf{return } (a\,,b)\}
\end{aligned} \tag{18}
$$

To understand this result, we can think of disintegration as "slicing" a measure. When we disintegrate a measure we slice it at the values of its observed variables (such as $x + \epsilon$), giving rise to a measure over the remaining variables (such as a and b).

Thus the result of calling **disintegrate** here is a function, and it represents the posterior measure over a and b given $x + \epsilon = t$. The input to the function is a real value t representing the data. The body of the function differs from the **blr** model in two ways. First, it is a distribution over a and b alone. Second, the bindings for x and ϵ are replaced by a **factor** expression that uses t.

In general, **disintegrate** replaces the binding for an observed variable with a **factor** involving the observed value. This factor comes from the probability mass assigned to the observed value by the likelihood (eq. (5) in our running example). We can interpret this factor as re-weighting the priors over a and b, i.e., updating our beliefs after observing data.

Disintegrations that satisfy the specification in eq. (13) represent posterior distributions [Chang and Pollard 1997; Shan and Ramsey 2017]. The **blr** example satisfies this specification:

$$
\begin{aligned}
&\textbf{do } \{t \leftsquigarrow \textbf{lebesgue};\\
&\quad p \leftsquigarrow \textbf{disintegrate blr } t;\\
&\quad \textbf{return } (t, p)\}\\
&\equiv \text{[eq. (18)]}\\
&\textbf{do } \{t \leftsquigarrow \textbf{lebesgue};\\
&\quad p \leftsquigarrow \textbf{do } \{a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\qquad\qquad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\qquad\qquad \textbf{factor } (\textbf{dGauss } 0\ 1\ (t - (a \cdot i + b)));\\
&\qquad\qquad \textbf{return } (a, b)\};\\
&\quad \textbf{return } (t, p)\}\\
&\equiv \text{[monad laws]}\\
&\textbf{do } \{t \leftsquigarrow \textbf{lebesgue};\\
&\quad a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\quad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\quad \textbf{factor } (\textbf{dGauss } 0\ 1\ (t - (a \cdot i + b)));\\
&\quad \textbf{return } (t, (a, b))\}\\
&\equiv [\textbf{dGauss } 0\ 1\ (x - y) \equiv \textbf{dGauss } y\ 1\ x\text{; definition of density in eq. (6)}]\\
&\textbf{do } \{a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\quad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\quad t \leftsquigarrow \textbf{normal } (a \cdot i + b)\ 1;\\
&\quad \textbf{return } (t, (a, b))\}\\
&\equiv \text{[eq. (55)]}\\
&\textbf{do } \{a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\quad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\quad x \leftsquigarrow \textbf{return } (a \cdot i + b);\\
&\quad \epsilon \leftsquigarrow \textbf{normal } 0\ 1;\\
&\quad \textbf{return } (x + \epsilon, (a, b))\}\\
&\equiv \textbf{blr}\ .
\end{aligned}
$$

This paper is about extending disintegration to handle arrays. To motivate and describe this extension, we first implement our version of scalar disintegration based on the work of Shan and Ramsey [2017].

3 IMPLEMENTING DISINTEGRATION

We now detail our implementation of Shan and Ramsey's disintegration technique. We implement disintegration as a program transformation whose input and output languages are both core Hakaru. Internally however, our disintegrator works with the extended language shown in fig. 4.

The goal of the disintegrator is to constrain the values of observed variables in the input program. We cannot always satisfy the constraints, and in such cases **disintegrate** fails to produce a result. Thus we design our implementation around recording dependencies and propagating constraints on bound variables.

The internal language extends core Hakaru with *heaps*, *bindings*, and *locations*. A *heap* is an ordered collection of *bindings*, where each binding is either a **factor** term or a *location* bound (via $\leftsquigarrow$)

$$
\begin{array}{lll}
\text{Real numbers } r \in \mathbb{R} & \text{Variables } x & \text{Locations } \bar{x} \\
\text{Bindings} & b ::= \bar{x} \leftsquigarrow m \mid \textbf{factor}\ e & \\
\text{Heap} & h ::= [b; \ldots; b] & \\
\text{Atomic terms} & u ::= x \mid -u \mid u + e \mid e + u \mid u \cdot e \mid e \cdot u \mid \textbf{fst}\ u \mid \textbf{snd}\ u & \\
 & \quad \mid \textbf{dGauss}\ u\ e\ e \mid \textbf{dGauss}\ e\ u\ e \mid \textbf{dGauss}\ e\ e\ u & \\
\text{Head normal forms} & v ::= u \mid r \mid \textbf{lebesgue} \mid \textbf{normal}\ e\ e \mid \textbf{return}\ e \mid \textbf{do}\ \{x \leftsquigarrow m; M\} & \\
 & \quad \mid \textbf{do}\ \{\textbf{factor}\ e; M\} \mid () \mid (e, e) & \\
\text{Terms} & e, m, M ::= v \mid \bar{x} \mid -e \mid e + e \mid e \cdot e \mid \textbf{dGauss}\ e\ e\ e \mid \textbf{fst}\ e \mid \textbf{snd}\ e & \\
\text{Types} & \alpha, \beta ::= \mathbb{1} \mid \mathbb{R} \mid \alpha \times \beta \mid \mathbb{M}\ \alpha &
\end{array}
$$

Fig. 4. The internal language of disintegration

to a measure expression. Locations (notated $\bar{x}$) are used by the disintegrator to track variables (notated x) originally bound in the input program. We order heaps with the oldest binding on the left, and the scope of a location follows this order.

To further streamline the recording and propagation of constraints, the internal language reorganizes terms into *atomic terms* and *head normal forms*. Atomic terms represent operations with at least one free variable. This makes observing atomic terms impossible, and disintegration (correctly) fails in these cases.

Disintegration is carried out mathematically by manipulating integrals [Chang and Pollard 1997; Shan and Ramsey 2017], and the disintegration of compound expressions (composed of multiple variables) is defined in terms of a change-of-variables operation. This operation involves differentiation with respect to one of the variables while keeping the others constant. To achieve this we need to evaluate the other variables to a *locally constant* head normal form. Thus we follow Shan and Ramsey [2017] in defining our disintegrator in terms of "forward" evaluating and "backward" constraining functions:

$$
\begin{array}{lll}
\gg\!\rhd \text{ (“perform”)} & : \mathit{heap} \to \lceil \mathbb{M}\ \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor) & (19) \\
\rhd \text{ (“evaluate”)} & : \mathit{heap} \to \lceil \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor) & (20) \\
\lll \text{ (“constrain outcome”)} & : \mathit{heap} \to \lceil \mathbb{M}\ \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap}) & (21) \\
\lhd \text{ (“constrain value”)} & : \mathit{heap} \to \lceil \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap}) & (22)
\end{array}
$$

The four functions are defined in fig. 5. The functions $\gg\!\rhd$ and $\rhd$ both partially evaluate [Fischer et al. 2011, 2008] their input terms (notated $\lceil \alpha \rceil$) to head normal form (notated $\lfloor \alpha \rfloor$). Now the disintegrator is non-effectful in that no random numbers are generated while producing the posterior program. Thus for $\gg\!\rhd$, evaluating measure terms to head normal form means simulating the act of making a random choice. We do this by *emitting* a binding for the measure to be evaluated, and returning a fresh variable that is atomic in the scope of the heap and the term during disintegration. The four functions of the disintegrator may add to but never look at this set of *emissions*.

Emissions occur in the $\gg\!\rhd$ cases of **lebesgue**, **normal**, and an atomic measure term u. On a **return** measure $\gg\!\rhd$ evaluates the point at which all probability mass is located.

Another role of $\gg\!\rhd$ is to record bindings onto the heap. To track a variable x bound by $x \leftsquigarrow m$, $\gg\!\rhd$ generates a location $\bar{x}$, stores the binding $\bar{x} \leftsquigarrow m$ on the heap, and replaces x with $\bar{x}$ within its scope. The heap is also used for storing **factor** expressions. In both cases, the disintegrator is *lazy* [Launchbury 1993] in that these expressions are stored without inspection or evaluation.

$\triangleright\!\!\triangleright$ ("perform") : $\mathit{heap} \to \lceil \mathbb{M}\ \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor)$

$\triangleright\!\!\triangleright\ h\ [\![m]\!] = \text{case } [\![m]\!] \text{ of}$

$[\![u]\!]$	$\to ([z \leftsquigarrow u], h, [\![z]\!])$	where u is atomic, z is fresh	(23)
$[\![\textbf{lebesgue}]\!]$	$\to ([z \leftsquigarrow \textbf{lebesgue}], h, [\![z]\!])$	where z is fresh	(24)
$[\![\textbf{normal}\ e\ e']\!]$	$\to ([s_1; s_2; z \leftsquigarrow \textbf{normal}\ v\ v'], h_2, [\![z]\!])$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \triangleright\ h_1\ [\![e']\!]$, z is fresh	(25)
$[\![\textbf{return}\ e]\!]$	$\to \triangleright\ h\ [\![e]\!]$		(26)
$[\![\textbf{do}\ \{x \leftsquigarrow e; M\}]\!]$	$\to \triangleright\!\!\triangleright\ [h; \bar{x} \leftsquigarrow e]\ [\![M]\!]\{x \mapsto \bar{x}\}$	where $\bar{x}$ is fresh	(27)
$[\![\textbf{do}\ \{\textbf{factor}\ e; M\}]\!]$	$\to \triangleright\!\!\triangleright\ [h; \textbf{factor}\ e]\ [\![M]\!]$		(28)
$[\![e]\!]$	$\to ([s_1; s_2], h_2, [\![v]\!])$	where e is not in head normal form, $(s_1, h_1, [\![m]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2, [\![v]\!]) = \triangleright\!\!\triangleright\ h_1\ [\![m]\!]$	(29)

$\triangleright$ ("evaluate") : $\mathit{heap} \to \lceil \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor)$

$\triangleright\ h\ [\![e_0]\!] = \text{case } [\![e_0]\!] \text{ of}$

$[\![v]\!]$	$\to ([], h, [\![v]\!])$	where v is in head normal form	(30)
$[\![\textbf{fst}\ e]\!]$	$\to$ case $[\![v]\!]$ of $[\![(e', \_)]\!] \to ([s_1; s_2], h_2, [\![v']\!])$ $[\![u]\!] \to (s_1, h_1, [\![\textbf{fst}\ u]\!])$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \triangleright\ h_1\ [\![e']\!]$, u is atomic	(31)
$[\![\textbf{snd}\ e]\!]$	$\to$ similar to the **fst** case		(32)
$[\![-e]\!]$	$\to (s, h_1, -[\![v]\!])$	where $(s, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$	(33)
$[\![e + e']\!]$	$\to ([s_1; s_2], h_2, [\![v]\!] + [\![v']\!])$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \triangleright\ h_1\ [\![e']\!]$	(34)
$[\![e \bullet e']\!]$	$\to ([s_1; s_2], h_2, [\![v]\!] \cdot [\![v']\!])$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \triangleright\ h_1\ [\![e']\!]$	(35)
$[\![\textbf{dGauss}\ \mu\ \sigma\ e]\!]$	$\to ([s_1; s_2; s_3], h_3, [\![\textbf{dGauss}\ c\ s\ v]\!])$	where $(s_1, h_1, [\![c]\!]) = \triangleright\ h\ [\![\mu]\!]$, $(s_2, h_2, [\![s]\!]) = \triangleright\ h_1\ [\![\sigma]\!]$, $(s_3, h_3, [\![v]\!]) = \triangleright\ h_2\ [\![e]\!]$	(36)
$[\![\bar{x}]\!]$	$\to (s, [h_1'; \bar{x} \leftsquigarrow \textbf{return}\ v; h_2], [\![v]\!])$	where $[h_1; \bar{x} \leftsquigarrow m; h_2] = h$, $(s, h_1', [\![v]\!]) = \triangleright\!\!\triangleright\ h_1\ [\![m]\!]$	(37)

$\lll$ ("constrain outcome") : $\mathit{heap} \to \lceil \mathbb{M}\ \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap})$

$\lll\ h\ [\![m]\!]\ t = \text{case } [\![m]\!] \text{ of}$

$[\![u]\!]$	$\to \bot$	where u is atomic	(38)
$[\![\textbf{lebesgue}]\!]$	$\to ([], h)$		(39)
$[\![\textbf{normal}\ e\ e']\!]$	$\to ([], [h; \textbf{factor}\ (\textbf{dGauss}\ e\ e'\ t)])$		(40)
$[\![\textbf{return}\ e]\!]$	$\to \lhd\ h\ [\![e]\!]\ t$		(41)
$[\![\textbf{do}\ \{x \leftsquigarrow e; M\}]\!]$	$\to \lll\ [h; \bar{x} \leftsquigarrow e]\ [\![M]\!]\{x \mapsto \bar{x}\}\ t$	where $\bar{x}$ is fresh	(42)
$[\![\textbf{do}\ \{\textbf{factor}\ e; M\}]\!]$	$\to \lll\ [h; \textbf{factor}\ e]\ [\![M]\!]\ t$		(43)
$[\![e]\!]$	$\to ([s_1; s_2], h_2)$	where e is not in head normal form, $(s_1, h_1, [\![m]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2) = \lll\ h_1\ [\![m]\!]\ t$	(44)

$\lhd$ ("constrain value") : $\mathit{heap} \to \lceil \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap})$

$\lhd\ h\ [\![e_0]\!]\ t = \text{case } [\![e_0]\!] \text{ of}$

$[\![\textbf{fst}\ e]\!]$	$\to$ case $[\![v]\!]$ of $[\![(e', \_)]\!] \to ([s_1; s_2], h_2)$ $[\![u]\!] \to \bot$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_2, h_2) = \lhd\ h_1\ [\![e']\!]\ t$, u is atomic	(45)
$[\![\textbf{snd}\ e]\!]$	$\to$ similar to the **fst** case		(46)
$[\![(e\ \textbf{,}\ e')]\!]$	$\to ([s_1; s_2], h_2)$	where $(s_1, h_1) = \lhd\ h\ [\![e]\!]\ (\text{fst}\ t)$, $(s_2, h_2) = \lhd\ h_1\ [\![e']\!]\ (\text{snd}\ t)$	(47)
$[\![-e]\!]$	$\to \lhd\ h\ [\![e]\!]\ (-t)$		(48)
$[\![e + e']\!]$	$\to ([s_1; s_1'], h_1') \sqcup ([s_2; s_2'], h_2')$	where $(s_1, h_1, [\![v]\!]) = \triangleright\ h\ [\![e]\!]$, $(s_1', h_1') = \lhd\ h_1\ [\![e']\!]\ (t - [\![v]\!])$, $(s_2, h_2, [\![v']\!]) = \triangleright\ h\ [\![e']\!]$, $(s_2', h_2') = \lhd\ h_2\ [\![e]\!]\ (t - [\![v']\!])$	(49)
$[\![v]\!]$	$\to \bot$	where v is in head normal form	(50)
$[\![\bar{x}]\!]$	$\to (s, [h_1'; \bar{x} \leftsquigarrow \textbf{return}\ t; h_2])$	where $[h_1; \bar{x} \leftsquigarrow m; h_2] = h$, $(s, h_1') = \lll\ h_1\ [\![m]\!]\ t$	(51)

Fig. 5. The implementation of our disintegrator

For most cases, we define $\triangleright$ as a standard partial evaluator. The cases for pair accessors **fst** and **snd** work by evaluating the argument and either recursively evaluating the sub-term or building a piece of code when the argument is atomic. For some operations that define atomic terms in fig. 4 (+, −, and ⋆), we use *smart constructors* that can simplify their arguments in special cases.

When the input term is a location $\bar{x}$, $\triangleright$ retrieves its binding from the heap and calls $\gg$ on the associated measure term. The monadic binding for $\bar{x}$ is modified into a deterministic let-binding that uses **return**, signalling the act of having made a random choice.

The functions $\ll$ and $\lhd$ constrain a term to be the observation t. Constraining a measure translates to computing the probability density (with respect to **lebesgue**) of that measure. Thus for a **normal** r_1 r_2 expression $\ll$ stores **factor** (**dGauss** r_1 r_2 t) on the heap, and for **lebesgue** it returns the heap unchanged. Just like $\gg$, $\ll$ also records monadic bind and **factor** expressions.

For a **return** term $\ll$ calls $\lhd$ on its point of mass. Many cases fail (with $\bot$) since **return** has no density with respect to **lebesgue** when the point mass is a constant. For the − case we constrain the value of the $\lhd$ to be $-t$, as we would when changing the variable of integration. In the case of + we could go forward on the first operand and constrain the second, or vice versa. This choice leads the disintegrator to provide two possible answers using $\sqcup$. In general, disintegration is non-deterministic in that it can produce multiple correct posterior distributions [Shan and Ramsey 2017]. In this paper we choose which output to describe based on simplicity.

On a location $\bar{x}$, $\lhd$ retrieves the heap binding and calls $\ll$ on the associated measure term. The location $\bar{x}$ is then let-bound to **return** t, signalling that this variable has been observed.

In summary, disintegration has three failure modes – when constraining an atomic term, when constraining a term that has been evaluated, and when constraining the same term multiple times. The implementation is designed to track these constraints.

3.1 Defining and Tracing an Algorithm

Given this implementation, we can describe an algorithm to perform conditioning:

$$\begin{aligned}
&\textbf{disintegrate} : \mathbb{M}\ (\alpha \times \beta) \to (\alpha \to \mathbb{M}\ \beta) \\
&\textbf{disintegrate}\ m\ t = \mathit{let}\ (e_1, h_1, v) = \gg\ []\ \ m \\
&\qquad\qquad\qquad\qquad\quad (e_2, h_2) \quad\ = \lhd\ h_1\ (\text{fst}\ v)\ t \\
&\qquad\qquad\qquad\qquad \mathit{in}\ \textbf{do}\ \{e_1; e_2; h_2; \textbf{return}\ (\text{snd}\ v)\}
\end{aligned} \tag{52}$$

The algorithm consists of three steps. Let us disintegrate **blr** and trace these steps:

(1) First we execute the input program m symbolically by calling $\gg$ [] m. This populates the heap with each monadic (and **factor**) binding that we encounter in the syntax tree until we reach the final line of the program (returning a pair). This is shown in fig. 6a.

(2) Next we condition on the first of the pair. Given the type of **disintegrate**, we expect that this is the term to be conditioned. We introduce the observation by invoking $\lhd$ h_1 (fst v) t. This is shown in fig. 6b.
We use the notation [——ǁ——] as a shorthand denoting the *previous* heap, i.e., we use this when the heap is unchanged between successive lines of the trace. The notation [——ǁ——; b] denotes the previous heap concatenated with the binding b.
In this step we observe that $x + \epsilon = t$, and we choose to first evaluate $\bar{x}$, as seen by the sub-call $\triangleright$ [...] [[$\bar{x}$]]. This in turn triggers evaluation of $\bar{a} \cdot i + \bar{b}$, and we emit bindings for a_z and b_z. We also indicate that $\bar{a}$ and $\bar{b}$ have been evaluated by updating them to let-bindings on the heap. This is informative to any procedures that might try later to constrain either of these locations. The result of evaluating $\bar{x}$ is thus $a_z \cdot i + b_z$.

We use this result to constrain the noise $\bar{\epsilon}$, as seen in the sub-call $\lhd$ [...] $[\![\bar{\epsilon}]\!]$ $(t - [\![a_z \cdot i + b_z]\!])$. This introduces a **factor** binding into the heap expressing the density of **normal** 0 1 at the observed value.

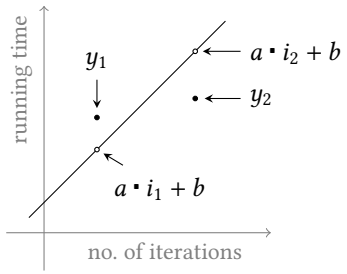
(3) Finally, we construct the posterior distribution by stitching together the emissions, final heap, and the second element of the pair:

$$\begin{aligned}
&\textbf{do } \{a_z \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\quad b_z \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\quad \bar{a} \leftsquigarrow \textbf{return } a_z;\\
&\quad \bar{b} \leftsquigarrow \textbf{return } b_z;\\
&\quad \bar{x} \leftsquigarrow \textbf{return } (a_z \cdot i + b_z);\\
&\quad \textbf{factor } (\textbf{dGauss } 0\ 1\ (t - (a_z \cdot i + b_z)));\\
&\quad \bar{\epsilon} \leftsquigarrow \textbf{return } (t - (a_z \cdot i + b_z));\\
&\quad \textbf{return } (\bar{a}, \bar{b})\}
\end{aligned} \tag{53}$$

In this last step we also convert locations back to variables, beta-reduce, and eliminate unused bindings to obtain the succinct core Hakaru output shown in eq. (18).

These three steps form the template for every disintegration computed with the functions in fig. 5. This includes cases involving tuples of data, which form an important intermediate step as we move towards more realistic examples involving arrays.

Making Two Measurements. We can develop the **blr** example to make *two* measurements of the running time of our algorithm. Consider an experiment where we noisily measure the running time first at i_1 and then at i_2 iterations:



$$\begin{aligned}
&\textbf{do } \{a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\
&\quad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\
&\quad y_1 \leftsquigarrow \textbf{normal } (a \cdot i_1 + b)\ 1;\\
&\quad y_2 \leftsquigarrow \textbf{normal } (a \cdot i_2 + b)\ 1;\\
&\quad \textbf{return } ((y_1, y_2), (a, b))\}
\end{aligned} \tag{54}$$

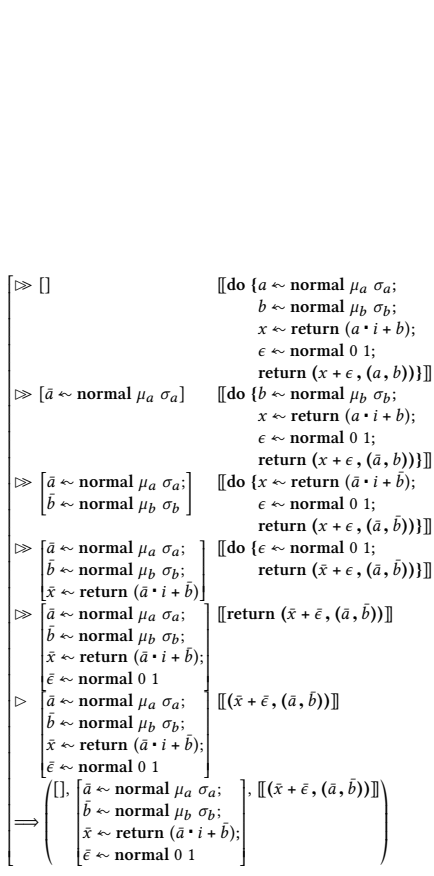
Here we model the slope a and intercept b as we did in eq. (1). For ease of exposition, we directly model the noisy measurements as y_1 and y_2, making use of the fact that:

$$\textbf{do } \{x \leftsquigarrow \textbf{return } (a \cdot i + b); \epsilon \leftsquigarrow \textbf{normal } 0\ 1; \textbf{return } (x + \epsilon)\} \equiv \textbf{normal } (a \cdot i + b)\ 1. \tag{55}$$

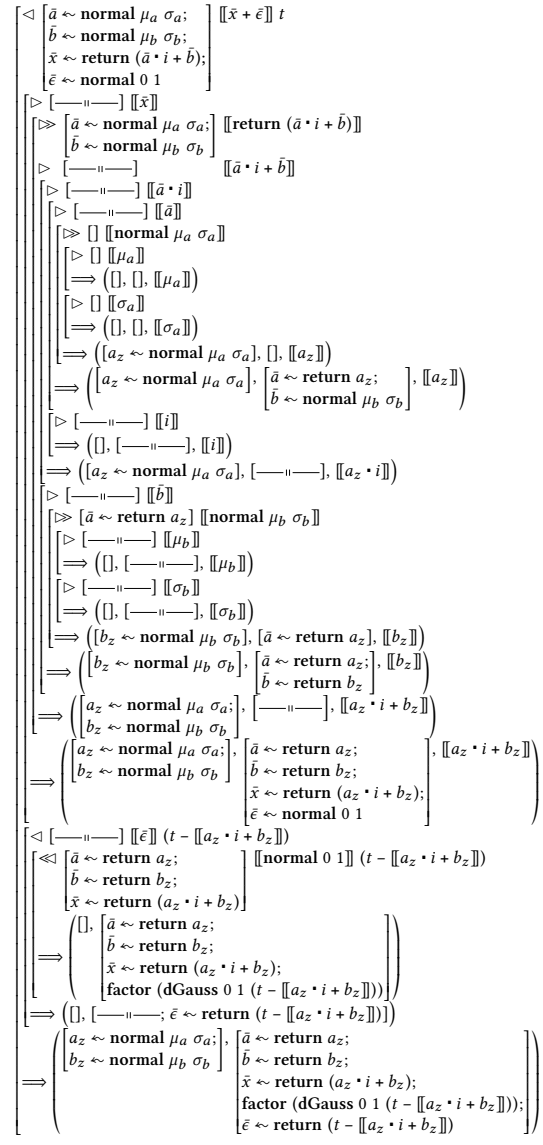
We can trace the algorithm as we did in the one-measurement case:

(1) We populate the heap as we traverse the term. This is shown in fig. 7a.

(2) We observe a pair of measurements by sequencing two observations: $\lhd$ [...] $[\![\bar{y}_1]\!]$ (**fst** t) and $\lhd$ [...] $[\![\bar{y}_2]\!]$ (**snd** t).
This is depicted in fig. 7b. This trace looks smaller than the one in fig. 6b, but only because we reformulated the model using the identity in eq. (55). Without this reformulation we would be dealing with $(x_1 + \epsilon_1)$ and $(x_2 + \epsilon_2)$ instead of y_1 and y_2, leading to roughly twice as many function calls as those in fig. 6b.



(a) Forward evaluation of the program

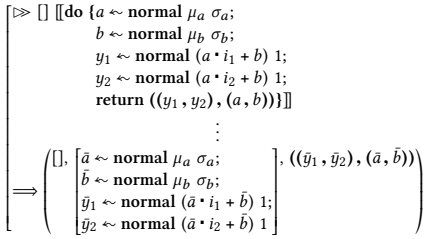


(b) Constraining the value of observed variable

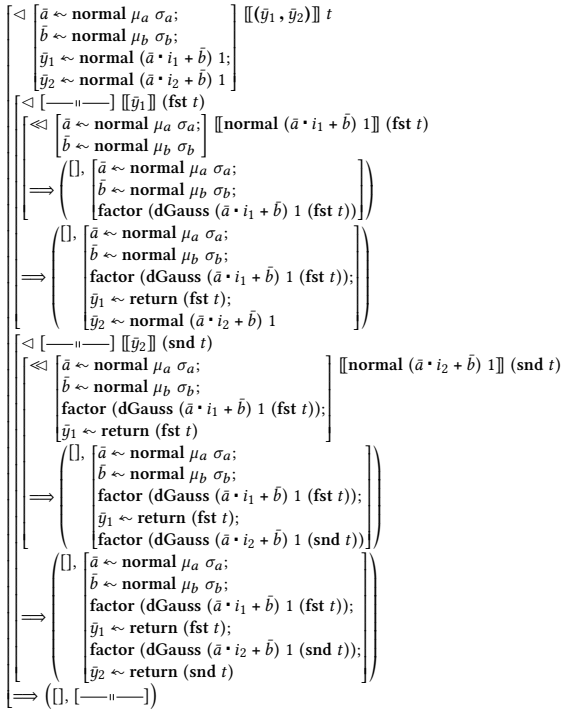
Fig. 6. The first two steps of disintegrating **blr**

(3) The final output (after beta-reduction and unused-binding-elimination) follows the same structure as before, containing one **factor** term for each noisy measurement:

$$\begin{aligned}&\textbf{do } \{a \leftsquigarrow \textbf{normal}\ \mu_a\ \sigma_a;\\ &\quad b \leftsquigarrow \textbf{normal}\ \mu_b\ \sigma_b;\\ &\quad \textbf{factor}\ (\textbf{dGauss}\ (a \cdot i_1 + b)\ 1\ (\textbf{fst}\ t));\\ &\quad \textbf{factor}\ (\textbf{dGauss}\ (a \cdot i_2 + b)\ 1\ (\textbf{snd}\ t));\\ &\quad \textbf{return}\ (a, b)\}\end{aligned} \tag{56}$$



(a) Going forward on the input program



(b) Constraining the value of the observed variable

Fig. 7. The first two steps of disintegrating two-measurement **blr**

Observing a variable removes its binding and multiplies a weight into the measure. In this case we observe two variables, and the total weight is a product of the weights occurring in the two **factor** expressions.

3.2 The Problem with Unrolling

As touched upon by the trace in fig. 7b, the disintegrator *unrolls* any structures that are observed in the input program. This creates inefficiencies that are exacerbated as the amount of data increases.

In the model in eq. (54) we construct a pair out of two separately bound noisy values; in a sense this input program is already unrolled. However, the output of the disintegrator would not change if we instead generated a pair of noisy measurements using a function (such as ⊗):

$$\begin{aligned}&\textbf{do } \{a \leftsquigarrow \textbf{normal } \mu_a\ \sigma_a;\\ &\quad b \leftsquigarrow \textbf{normal } \mu_b\ \sigma_b;\\ &\quad y \leftsquigarrow (\textbf{normal } (a \bullet i_1 + b)\ 1) \otimes (\lambda\_.\ \textbf{normal } (a \bullet i_2 + b)\ 1);\\ &\quad \textbf{return } (y, (a, b))\}\end{aligned} \tag{57}$$

The disintegrator in fig. 5 would produce a result identical to the one seen in eq. (56)—the pair of noisy values would be unrolled and the output program would be flattened in relation to the input model.

Ideally we'd make several measurements to estimate the slope and intercept in **blr**. We might use tuples or arrays of arbitrary length to store these measurements, and the earlier method of

unrolling the pair structure will have to be generalized to these arbitrary-length structures. Such an approach is conceptually sound; systems such as PSI [Gehr et al. 2016] unroll their arrays.

We could encode arrays in core Hakaru using nested tuples: $(x + \epsilon_1\,,\,(x + \epsilon_2\,,\,(x + \epsilon_3\,,\,\ldots)))$. The disintegrator in fig. 5 can already handle such nested tuples, but producing a disintegration will take time and space linear in the size of the input, i.e., in the number of scalar elements in the array. This is not desirable, considering that we expect to condition our models on large quantities of data.

The problem seems especially grim when we consider deeper structures, i.e., when we move from singly-indexed to multiply-indexed arrays. Whereas an array of n numbers takes $O(n)$ time and space to disintegrate, an array of n arrays each of n numbers takes $O(n^2)$ time and space. In a modular approach to inference, this increase in the size of the output program would adversely affect any transformation that we may wish to use downstream from **disintegrate**.

This motivates a new technique for disintegrating probabilistic programs that contain arrays of stochastic variables, one that manipulates arrays symbolically and without unrolling. First, we extend the language of programs to allow the expression and manipulation of arrays.

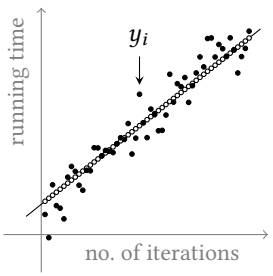
4 AN ARRAY PROBABILISTIC LANGUAGE

We extend core Hakaru with constructs for writing array probabilistic programs. Fig. 8 describes this extended language.

$$
\begin{array}{lll}
\text{Real numbers } r \in \mathbb{R} & \text{Natural numbers } n \in \mathbb{N} & \text{Variables } x \\
\text{Terms } e, l, m, M ::= x \mid r \mid n \mid -e \mid e + e \mid e \cdot e \mid \textbf{dGauss}\ e\ e\ e \mid e = e \mid () \mid (e\,,\,e) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\
\qquad \mid \textbf{lebesgue} \mid \textbf{normal}\ e\ e \mid \textbf{return}\ e \mid \textbf{do}\ \{x \leftsquigarrow m;\ M\} \mid \textbf{do}\ \{\textbf{factor}\ e;\ M\} \\
\qquad \mid \textbf{counting} \mid \textbf{array}\ l\ x.\ e \mid \textbf{idx}\ e\ e \mid \textbf{plate}\ l\ x.\ m \mid \textbf{size}\ e \\
\text{Types} \quad \alpha, \beta ::= \mathbb{1} \mid \mathbb{R} \mid \mathbb{N} \mid \alpha \times \beta \mid \langle \alpha \rangle \mid \mathbb{M}\ \alpha
\end{array}
$$

Fig. 8. Core Hakaru extended with =, **counting**, **array**, **plate**, **idx**, and **size**

We can understand this language by example. Consider that we update **blr** to make k noisy measurements of the running time at evenly-spaced iterations. Concretely, imagine that we measure the running times at 100 iterations, then 200, 300, and so on until we reach $k \times 100$ iterations:



$$
\begin{array}{l}
\textbf{do}\ \{a \leftsquigarrow \textbf{normal}\ \mu_a\ \sigma_a; \\
\quad b \leftsquigarrow \textbf{normal}\ \mu_b\ \sigma_b; \\
\quad y \leftsquigarrow \textbf{plate}\ k\ i. \\
\qquad\quad \textbf{normal}\ (a \cdot 100 \cdot i + b)\ 1; \\
\quad \textbf{return}\ (y\,,\,(a\,,\,b))\}
\end{array}
\tag{58}
$$

In this model we use a new construct: **plate**. This is a primitive used for expressing loops containing stochastic computation. The construct derives its name from "plate notation", a shorthand used in the literature on graphical models and Bayesian inference [as reviewed in Buntine 1994], representing variables that repeat.

The output of **plate** is an array that collects the result of each loop iteration. The first argument is the size of this array, i.e., the number of loop iterations. The second argument is a variable (of type $\mathbb{N}$) representing an index for each iteration. This variable takes scope over the third argument, which represents the body of the loop. The body has measure type, indicating that each iteration is a stochastic computation.

In the model in eq. (58), we use **plate** to draw a noisy measurement of each of the running times from 100 iterations to $k \times 100$ iterations. These k measurements are stored into the array y that is bound as the result of calling **plate**.

The language in fig. 8 introduces a few additional array constructs. Fig. 9 shows the typing rules for these primitives; we discuss them further below.

$$\frac{}{\Gamma \vdash \mathbf{counting} : \mathbb{M}\,\mathbb{N}} \qquad \frac{\Gamma \vdash l : \mathbb{N} \quad \Gamma, x{:}\mathbb{N} \vdash m : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{plate}\ l\ x.\ m : \mathbb{M}\,\langle\alpha\rangle} \qquad \frac{\Gamma \vdash l : \mathbb{N} \quad \Gamma, x{:}\mathbb{N} \vdash e : \alpha}{\Gamma \vdash \mathbf{array}\ l\ x.\ e : \langle\alpha\rangle}$$

$$\frac{\Gamma \vdash e : \langle\alpha\rangle \quad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash \mathbf{idx}\ e\ e' : \alpha} \qquad \frac{\Gamma \vdash e : \langle\alpha\rangle}{\Gamma \vdash \mathbf{size}\ e : \mathbb{N}} \qquad \frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash e = e' : \mathbb{N}}$$

Fig. 9. Typing rules for the extensions to core Hakaru

The **array** constructor is used for generating an array. The first two arguments are identical to those for the **plate** primitive. The third argument is an expression that describes each element of the array. For example, **array** 100 x. x defines an array of the first 100 natural numbers.

The remaining additions to core Hakaru are straightforward. The set of types now includes the natural numbers $\mathbb{N}$ and the type of arrays, notated $\langle\alpha\rangle$. Terms can contain natural numbers n, while the addition and negation operations are overloaded to work on both real and natural numbers. Furthermore, we assume that natural numbers can be implicitly promoted to reals when necessary. The **counting** primitive denotes the unique translation-invariant measure over $\mathbb{N}$. While the **size** primitive obtains the length of an array, the **idx** primitive indexes into an array at a given expression of type $\mathbb{N}$. Finally, the = primitive returns 1 if two expressions represent the same natural number, and 0 otherwise.

5 DISINTEGRATING PROGRAMS WITH ARRAYS

Given a probabilistic program written in the language of fig. 8, we want to extend **disintegrate** to automatically produce the posterior distribution in the same language. For the model in eq. (58), we want to produce:

$$\begin{aligned}
&\lambda t.\ \mathbf{do}\ \{a \leftsquigarrow \mathbf{normal}\ \mu_a\ \sigma_a;\\
&\qquad b \leftsquigarrow \mathbf{normal}\ \mu_b\ \sigma_b;\\
&\qquad _ \leftsquigarrow \mathbf{plate}\ k\ i.\ \mathbf{do}\ \{\mathbf{factor}\ (\mathbf{dGauss}\ (a \cdot 100 \cdot i + b)\ 1\ (\mathbf{idx}\ t\ i));\\
&\qquad\qquad\qquad \mathbf{return}\ ()\};\\
&\qquad \mathbf{factor}\ (k = \mathbf{size}\ t);\\
&\qquad \mathbf{return}\ (a, b)\}
\end{aligned} \tag{59}$$

This is a conditional measure where the input t is an array of real values. This output generalizes the result in eq. (56) from a having a product of 2 to a product of k **factor**s, each of which is derived from the density of the Gaussian distribution representing a noisy measurement. To set up disintegrate to produce this symbolic product, we extend the language of fig. 8.

Real numbers $r \in \mathbb{R}$ Natural numbers $n \in \mathbb{N}$ Variables x Locations $\bar{x}$ Indices $\hat{x}$

$$
\begin{array}{lll}
\text{Bindings} & b & ::= \bar{x} \leftsquigarrow [\hat{x}{\cdot}l{\ldots}].\, m \mid \mathbf{factor}\ [\hat{x}{\cdot}l{\ldots}].\, e \\
\text{Heap} & h & ::= [b; \ldots; b] \\
\text{Atomic terms} & u & ::= x \mid -u \mid u + e \mid e + u \mid u \cdot e \mid e \cdot u \mid \mathbf{fst}\ u \mid \mathbf{snd}\ u \mid \mathbf{idx}\ u\ e \mid \mathbf{idx}\ e\ u \mid \mathbf{size}\ u \\
 & & \quad \mid \mathbf{dGauss}\ u\ e\ e \mid \mathbf{dGauss}\ e\ u\ e \mid \mathbf{dGauss}\ e\ e\ u \mid u = e \mid e = u \\
\text{Head normal forms} & v & ::= u \mid r \mid n \mid () \mid (e, e) \mid \mathbf{counting} \mid \mathbf{lebesgue} \mid \mathbf{normal}\ e\ e \mid \mathbf{return}\ e \\
 & & \quad \mid \mathbf{do}\ \{x \leftsquigarrow m; M\} \mid \mathbf{do}\ \{\mathbf{factor}\ e; M\} \mid \mathbf{plate}\ l\ x.\ m \mid \mathbf{array}\ l\ x.\ e \\
\text{Terms} & e, l, m, M & ::= v \mid \bar{x}[\hat{x}{\ldots}] \mid \langle \bar{x}[\hat{x}{\ldots}] \rangle \mid \hat{x} \mid -e \mid e + e \mid e \cdot e \mid \mathbf{dGauss}\ e\ e\ e \mid e = e \\
 & & \quad \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{idx}\ e\ e \mid \mathbf{size}\ e \\
\text{Types} & \alpha, \beta & ::= \mathbb{1} \mid \mathbb{R} \mid \mathbb{N} \mid \alpha \times \beta \mid \langle \alpha \rangle \mid \mathbb{M}\ \alpha
\end{array}
$$

Fig. 10. Internal language for disintegrating array programs

5.1 A Language of Repeated Bindings

Fig. 10 shows the internal language used for disintegrating array probabilistic programs. As in the case of the previous internal language (of fig. 4), we track bound variables by extending the input language with a heap of bindings. When a binding originates under an array, it would in principle be repeated many times on the heap. To express this repetition succinctly, the internal language introduces *index variables*.

Index variables, or *indices* for short, are variables of type $\mathbb{N}$ that are used for tracking repeated expressions. Notated $\hat{x}$, they are created by the disintegrator when it goes under the binder of **array** and **plate** expressions. Just as a location $\bar{x}$ represents the variable bound from $x \leftsquigarrow e$, an index $\hat{x}$ represents the bound variable in **array** $l\ x.\ e$ or **plate** $l\ x.\ e$.

While a location is bound on the heap, an index is bound by a list notated $[\hat{x}{\cdot}l{\ldots}]$. Here each element of the list is an index variable paired with an expression of type $\mathbb{N}$ that denotes the length of the corresponding array. Since the language permits arbitrary nesting of arrays, the list can contain zero or more elements. The ordering of the elements matters here, and the list—being a binding form—contains no repetitions. We label this structure a *pairwise-distinct list*, and refer to its instances as *binding-lists*.

The binding-list represents repetition. An expression together with a binding-list such as the pairwise-distinct list $[\hat{x}_1 \cdot l_1, \ldots, \hat{x}_n \cdot l_n]$ compactly represents that expression repeated a total of $l_1 \cdots l_n$ times, with any uses of the variables $\hat{x}_1, \ldots, \hat{x}_n$ updated accordingly. When an expression is augmented with a binding-list we deem the expression *lifted*.

Every binding is now lifted. This means that locations and weights are (conceptually) repeated zero or more times on the heap. When a location is used, it must be "applied" to a list of indices to select from all the symbolic bindings under that location. This list—notated $[\hat{x}{\ldots}]$—contains indices alone, and is akin to an array *accessor* or *selector*. We refer to its instances as *selector-lists*.

The scoping rules for locations on the heap remain unchanged from the previous disintegrator—older bindings are to the left and take scope over younger bindings to the right. Binding-lists, on the other hand, take scope only over the rest of the *current* binding. This is better illustrated via an example heap:

$$\left[\bar{a} \leftsquigarrow [].\, \mathbf{normal}\ 0\ 1;\ \bar{b} \leftsquigarrow [\hat{x}{\cdot}50].\, \mathbf{normal}\ \bar{a}[]\ 1;\ \mathbf{factor}\ [\hat{x}{\cdot}30].\, (\bar{b}[\hat{x}] + \hat{x})\right] \tag{60}$$

The $\bar{b}$ in $\bar{b}[\hat{x}] + \hat{x}$ refers to the binding $\bar{b} \leftsquigarrow [\hat{x}{\cdot}50]$. **normal** $\bar{a}[]$ 1. On the other hand, the $\hat{x}$ in $\bar{b}[\hat{x}] + \hat{x}$ refers to $\hat{x}{\cdot}30$ and not $\hat{x}{\cdot}50$.

Just as conditioning on a pair of terms involves disintegrating each subterm, conditioning on an array involves successive disintegration on each array element. Thus, the four functions of disintegration are themselves lifted:

$$\gg\!\triangleright \text{ (“perform”)} \quad : \mathit{heap} \to \mathit{inds} \to \lceil \mathbb{M}\ \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor) \tag{61}$$

$$\triangleright \text{ (“evaluate”)} \quad : \mathit{heap} \to \mathit{inds} \to \lceil \alpha \rceil \to (\mathit{emission} \times \mathit{heap} \times \lfloor \alpha \rfloor) \tag{62}$$

$$\lessdot \text{ (“constrain outcome”)} : \mathit{heap} \to \mathit{inds} \to \lceil \mathbb{M}\ \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap}) \tag{63}$$

$$\triangleleft \text{ (“constrain value”)} \quad : \mathit{heap} \to \mathit{inds} \to \lceil \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap}) \tag{64}$$

The binding-lists here take scope over the rest of the arguments—the term being disintegrated and the observation (when present). This captures any indices and the selector-lists of locations used within the term or the observation.

5.2 Extending the Previous Disintegrator

Figs. 11 and 12 illustrate the four functions implementing lifted disintegration, which we describe in more detail here. Most of the extensions are straightforward.

Where earlier we may have emitted a binding—as in the case of $\gg\!\triangleright$ [...] [[**lebesgue**]]—we may now emit a **plate** binding. The number of nested **plate**s, which may be zero, is determined by the binding-list $[\hat{x}{\cdot}l ...]$ of the current function. We notate this using **plate** $l\ x.\ \ldots e$ syntax. As an example, $\gg\!\triangleright$ $[\hat{x}{\cdot}l, \hat{y}{\cdot}l']$ [...] [[**lebesgue**]] emits a fresh binding for **plate** $l\ x.$ (**plate** $l'\ y.$ **lebesgue**).

Where earlier we may have stored a binding on the heap—as in the case of performing a monadic expression, i.e., $\gg\!\triangleright$ [...] [[**do** $\{x \leftsquigarrow e;\ M\}$]]—we now store a lifted binding. In these cases we lift the bindings with the binding-list of the current function. In these cases we also generate a fresh location, and all bindings of that location are selected for use, i.e, we apply each use of that location to a list of all indices in the current binding-list.

Thus, in most cases the lifting binding-list remains unchanged. We can now consider one-by-one the cases that use the binding-list non-trivially.

The **plate** constructor is a measure operation that encapsulates repeated measure terms. For this case $\gg\!\triangleright$ and $\lessdot$ both store on the heap a binding for e in **plate** $l\ x.\ e$. Since **plate** itself repeats e l times, the binding is lifted by the current binding-list *extended* with a fresh element $\hat{x}{\cdot}l$. Since **plate** returns an array, $\gg\!\triangleright$ and $\lessdot$ need to recur on all l bindings in an array form. For this purpose we use the *multiloc* constructor $\langle \bar{x}[\hat{x} ...] \rangle$, which denotes an array of locations. It is applied to a selector-list whose length is one less than the size of the binding-list for the corresponding location bound on the heap.

The **idx** operator generalizes **fst** and **snd** to the array case. Thus when either operand of **idx** $e_1\ \ e_2$ is atomic, the result looks similar to handling pair destructors on an atomic operand: $\triangleright$ returns code while $\triangleleft$ fails. When the index operand (e_2) is an index variable, we symbolically access an element of the array operand e_1. For an **array** expression this involves substitution of the index variable into the body of the array. For a *multiloc* expression we extend its selector-list to select the corresponding location. Both $\triangleright$ and $\triangleleft$ recur on the newly accessed term. We note that neither function currently handles the case of indexing into an array at non-atomic and non-index operands such as literal numbers.

The **array** and multiloc constructors are head normal forms; $\triangleright$ does not treat them specially. For $\triangleleft$ the case for pairs is informative—we go under the structure and make successive calls to $\triangleleft$ for each element. We repeat disintegration by extending the binding-list for $\triangleleft$ with a fresh index

$\gg$ ("perform") : $heap \to inds \to \lceil \mathbb{M}\ \alpha \rceil \to (emission \times heap \times \lfloor \alpha \rfloor)$

$\gg\ h\ [\hat{x}\cdot l\ldots]\ [\![m]\!] = \text{case } [\![m]\!] \text{ of}$

$[\![u]\!]$	$\to ([z \leftsquigarrow \textbf{plate}\ l\ x.\ \ldots\ u], h, [\![z]\!])$	where u is atomic, z is fresh	(65)
$[\![\textbf{lebesgue}]\!]$	$\to ([z \leftsquigarrow \textbf{plate}\ l\ x.\ \ldots \textbf{lebesgue}], h, [\![z]\!])$	where z is fresh	(66)
$[\![\textbf{normal}\ e\ e']\!]$	$\to ([s_1; s_2; z \leftsquigarrow \textbf{plate}\ l\ x.\ \ldots \textbf{normal}\ e\ e'], h_2, [\![z]\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$, z is fresh	(67)
$[\![\textbf{return}\ e]\!]$	$\to \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$		(68)
$[\![\textbf{do}\ \{x \leftsquigarrow e; M\}]\!]$	$\to \gg\ [h; \bar{x} \leftsquigarrow [\hat{x}\cdot l\ldots].\ e]\ [\hat{x}\cdot l\ldots]\ [\![M]\!]\{x \mapsto \bar{x}[\hat{x}\ldots]\}$	where $\bar{x}$ is fresh	(69)
$[\![\textbf{do}\ \{\textbf{factor}\ e; M\}]\!]$	$\to \gg\ [h; \textbf{factor}\ [\hat{x}\cdot l\ldots].\ e]\ [\hat{x}\cdot l\ldots]\ [\![M]\!]$		(70)
$[\![\textbf{plate}\ l'\ y.\ e]\!]$	$\to \rhd\ [h; \bar{p} \leftsquigarrow [\hat{x}\cdot l\ldots\hat{y}\cdot l'].\ e\{y \mapsto \hat{y}\}]\ [\hat{x}\cdot l\ldots]\ \langle \bar{p}[\hat{x}\ldots]\rangle$	where $\bar{p}, \hat{y}$ are fresh	(71)
$[\![e]\!]$	$\to ([s_1; s_2], h_2, [\![v]\!])$	where e is not in head normal form, $(s_1, h_1, [\![m]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v]\!]) = \gg\ h_1\ [\hat{x}\cdot l\ldots]\ [\![m]\!]$	(72)

$\rhd$ ("evaluate") : $heap \to inds \to \lceil \alpha \rceil \to (emission \times heap \times \lfloor \alpha \rfloor)$

$\rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e_0]\!] = \text{case } [\![e_0]\!] \text{ of}$

$[\![v]\!]$	$\to ([], h, [\![v]\!])$	where v is in head normal form	(73)
$[\![\textbf{fst}\ e]\!]$	$\to$ case $[\![v]\!]$ of $[\![(e', \_)]\!] \to ([s_1; s_2], h_2, [\![v']\!])$ $[\![u]\!] \to (s_1, h_1, [\![\textbf{fst}\ u]\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$, u is atomic	(74)
$[\![\textbf{snd}\ e]\!]$	$\to$ similar to the **fst** case		(75)
$[\![\textbf{idx}\ e\ e']\!]$	$\to$ if v is atomic then $(s_1, h_1, [\![\textbf{idx}\ v\ e']\!])$ else if v' is atomic then $([s_1; s_2], h_2, [\![\textbf{idx}\ v\ v']\!])$ else case $[\![(v, v')]\!]$ of $[\![(\textbf{array}\ l'\ y.\ a, \hat{y})]\!] \to ([s_1; s_2; s_3], h_3, [\![v'']\!])$ $[\![(\langle \bar{x}[\hat{y}\ldots]\rangle, \hat{z})]\!] \to ([s_1; s_2; s_3], h_3, [\![v'']\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$ $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$ $(s_3, h_3, [\![v'']\!]) = \rhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![a]\!]\{y \mapsto \hat{y}\}$ $(s_3, h_3, [\![v'']\!]) = \rhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![\bar{x}[\hat{y}\ldots\hat{z}]]\!]$	(76)
$[\![\textbf{size}\ e]\!]$	$\to (s, h_1, \text{size}\ [\![v]\!])$	where $(s, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$	(77)
$[\![-e]\!]$	$\to (s, h_1, -[\![v]\!])$	where $(s, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$	(78)
$[\![e + e']\!]$	$\to ([s_1; s_2], h_2, [\![v]\!] + [\![v']\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$	(79)
$[\![e \bullet e']\!]$	$\to ([s_1; s_2], h_2, [\![v]\!] \cdot [\![v']\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$	(80)
$[\![\textbf{dGauss}\ e\ e'\ e'']\!]$	$\to ([s_1; s_2; s_3], h_3, [\![\textbf{dGauss}\ v\ v'\ v'']\!])$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$, $(s_3, h_3, [\![v'']\!]) = \rhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![e'']\!]$	(81)
$[\![\bar{x}[\hat{y}\ldots]]\!]$	$\to (s, [h'_1; \bar{x} \leftsquigarrow [\hat{z}\cdot l'\ldots].\ \textbf{return}\ v; h_2], [\![v]\!]\{[\hat{z}\ldots] \mapsto [\hat{y}\ldots]\})$	where $[h_1; \bar{x} \leftsquigarrow [\hat{z}\cdot l'\ldots].\ m; h_2] = h$, $\lvert[\hat{y}\ldots]\rvert = \lvert[\hat{z}\cdot l'\ldots]\rvert$, $(s, h'_1, [\![v]\!]) = \gg\ h_1\ [\hat{z}\cdot l'\ldots]\ [\![m]\!]$	(82)

Fig. 11. The forward functions handling arrays

variable. In the **array** case we substitute this index under the binder and call $\lhd$ on the body. In the multiloc case we select a location by extending its selector-list, and we call $\lhd$ on the resulting term.

Constraining the **array** and multiloc constructors repeatedly constrains each element to be the corresponding element of the observed variable t (obtained using **idx**). While t is guaranteed by the type-system to be an array, we introduce a guard that constrains the size of t. This size check appears in the output in eq. (59). To see the need for this constraint we consider the new specification of **disintegrate**.

$\ll$ ("constrain outcome") : $\mathit{heap} \to \mathit{inds} \to \lceil \mathbb{M}\ \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap})$

$\ll\ h\ [\hat{x}\cdot l\ldots]\ [\![m]\!]\ t = \mathbf{case}\ [\![m]\!]\ \mathbf{of}$

$[\![u]\!]$	$\to \bot$	where u is atomic	(83)
$[\![\mathbf{lebesgue}]\!]$	$\to ([\,], h)$		(84)
$[\![\mathbf{normal}\ e\ e']\!]$	$\to ([\,], [h; \mathbf{factor}\ [\hat{x}\cdot l\ldots].\ \mathbf{dGauss}\ e\ e'\ t])$		(85)
$[\![\mathbf{return}\ e]\!]$	$\to \lhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]\ t$		(86)
$[\![\mathbf{do}\ \{x \leftsquigarrow e; M\}]\!]$	$\to \ll [h; \bar{x} \leftsquigarrow [\hat{x}\cdot l\ldots].\ e]\ [\hat{x}\cdot l\ldots]\ [\![M]\!]\{x \mapsto \bar{x}[\hat{x}\ldots]\}\ t$	where $\bar{x}$ is fresh	(87)
$[\![\mathbf{do}\ \{\mathbf{factor}\ e; M\}]\!]$	$\to \ll [h; \mathbf{factor}\ [\hat{x}\cdot l\ldots].\ e]\ [\hat{x}\cdot l\ldots]\ [\![M]\!]\ t$		(88)
$[\![\mathbf{plate}\ l'\ y.\ e]\!]$	$\to \lhd\ [h; \bar{p} \leftsquigarrow [\hat{x}\cdot l\ldots\hat{y}\cdot l'].\ e\{y \mapsto \hat{y}\}]\ [\hat{x}\cdot l\ldots]\ [\![\langle \bar{p}[\hat{x}\ldots]\rangle]\!]\ t$	where $\bar{p}$, $\hat{y}$ are fresh	(89)
$[\![e]\!]$	$\to ([s_1; s_2], h_2)$	where e is not in head normal form, $(s_1, h_1, [\![m]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2) = \ll h_1\ [\hat{x}\cdot l\ldots]\ [\![m]\!]\ t$	(90)

$\lhd$ ("constrain value") : $\mathit{heap} \to \mathit{inds} \to \lceil \alpha \rceil \to \lfloor \alpha \rfloor \to (\mathit{emission} \times \mathit{heap})$

$\lhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e_0]\!]\ t = \mathbf{case}\ [\![e_0]\!]\ \mathbf{of}$

$[\![\mathbf{fst}\ e]\!]$	$\to$ case $[\![v]\!]$ of $[\![(e', \_)]\!] \to ([s_1; s_2], h_2)$ $[\![u]\!] \to \bot$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_2, h_2) = \lhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]\ t$, u is atomic	(91)
$[\![\mathbf{snd}\ e]\!]$	$\to$ similar to the **fst** case		(92)
$[\![\mathbf{idx}\ e\ e']\!]$	$\to$ if v is atomic then $\bot$ else if v' is atomic then $\bot$ else case $[\![(v, v')]\!]$ of $[\![(\mathbf{array}\ l'\ y.\ a, \hat{y})]\!] \to ([s_1; s_2; s_3], h_3)$ $[\![(\langle \bar{x}[\hat{y}\ldots]\rangle, \hat{z})]\!] \to ([s_1; s_2; s_3], h_3)$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$ $(s_2, h_2, [\![v']\!]) = \rhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$ $(s_3, h_3) = \lhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![a]\!]\{y \mapsto \hat{y}\}\ t$ $(s_3, h_3) = \lhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![\bar{x}[\hat{y}\ldots\hat{z}]]\!]\ t$	(93)
$[\![()]\!]$	$\to ([\,], h)$		(94)
$[\![(e, e')]\!]$	$\to ([s_1; s_2], h_2)$	where $(s_1, h_1) = \lhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]\ (\mathrm{fst}\ t)$, $(s_2, h_2) = \lhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]\ (\mathrm{snd}\ t)$	(95)
$[\![\mathbf{array}\ l'\ y.\ e]\!]$	$\to \lhd\ [h; \mathbf{factor}\ [\hat{x}\cdot l\ldots].\ l' = \mathbf{size}\ t]\ [\hat{x}\cdot l\ldots\hat{y}\cdot l']\ [\![e]\!]\{y \mapsto \hat{y}\}\ (\mathbf{idx}\ t\ \hat{y})$ where $\hat{y}$ is fresh		(96)
$[\![\langle \bar{x}[\hat{y}\ldots]\rangle]\!]$	$\to \lhd\ [h; \mathbf{factor}\ [\hat{x}\cdot l\ldots].\ l'' = \mathbf{size}\ t]\ [\hat{x}\cdot l\ldots\hat{q}\cdot l'']\ [\![\bar{x}[\hat{y}\ldots\hat{q}]]\!]\ (\mathbf{idx}\ t\ \hat{q})$ where $[h_1; \bar{x} \leftsquigarrow [\hat{z}\cdot l'\ldots\hat{p}\cdot l'']\ .\ m; h_2]$, $\hat{q}$ is fresh, $\lvert[\hat{y}\ldots]\rvert = \lvert[\hat{z}\cdot l'\ldots\hat{p}\cdot l'']\rvert - 1$,		(97)
$[\![-e]\!]$	$\to \lhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]\ (-t)$		(98)
$[\![e + e']\!]$	$\to ([s_1; s_1'], h_1')$ $\sqcup\ ([s_2; s_2'], h_2')$	where $(s_1, h_1, [\![v]\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e]\!]$, $(s_1', h_1') = \lhd\ h_1\ [\hat{x}\cdot l\ldots]\ [\![e']\!]\ (t - [\![v]\!])$ where $(s_2, h_2, [\![v']\!]) = \rhd\ h\ [\hat{x}\cdot l\ldots]\ [\![e']\!]$, $(s_2', h_2') = \lhd\ h_2\ [\hat{x}\cdot l\ldots]\ [\![e]\!]\ (t - [\![v']\!])$	(99)
$[\![v]\!]$	$\to \bot$	where v is in head normal form	(100)
$[\![\bar{x}[\hat{y}\ldots]]\!]$	$\to (s, [h_1'; \bar{x} \leftsquigarrow [\hat{x}\cdot l\ldots].\ \mathbf{return}\ t; h_2])$	where $[h_1; \bar{x} \leftsquigarrow [\hat{z}\cdot l'\ldots].\ m; h_2] = h$, $\lvert[\hat{y}\ldots]\rvert = \lvert[\hat{z}\cdot l'\ldots]\rvert$, $[\hat{y}\ldots]$ is a permutation of $[\hat{x}\ldots]$, $(s, h_1') = \ll h_1\ [\hat{x}\cdot l\ldots]\ [\![m]\!]\{[\hat{z}\ldots] \mapsto [\hat{y}\ldots]\}\ t$	(101)

Fig. 12. The backward functions handling arrays

Specifying Disintegration on Arrays. The specification of disintegration depends on the type of the observed variable. Previously we specified disintegration when the input data is a real number, where **lebesgue** is used as the base measure. When observing pairs of real numbers, we update our specification and disintegrate with respect to **lebesgue** $\otimes$ ($\lambda\_.$ **lebesgue**). When observing arrays

of real numbers, the specification becomes:

$$\textbf{do } \{n \leftsquigarrow \textbf{counting}; t \leftsquigarrow \textbf{plate } n \ \_. \ \textbf{lebesgue}; p \leftsquigarrow \textbf{disintegrate } m \ t; \textbf{return } (t, p)\} \equiv m \tag{102}$$

Here the sub-expression **do** $\{n \leftsquigarrow$ **counting**; **plate** n _. **lebesgue**$\}$ actually defines the *disjoint union* of Lebesgue measures on $\mathbb{R}^n$ for each value of n. To select the correct base measure from this disjoint union, the output of **disintegrate** must apply a guard checking the size of the input array t and the observed variable inside the model.

Now only the cases for locations remain to be described. When handling locations the disintegrator has to compare three quantities—the binding-list $[\hat{x}{\cdot}l\ldots]$ of the function call, the binding-list $[\hat{z}{\cdot}l'\ldots]$ lifting this location's binding on the heap, and the selector-list $[\hat{y}\ldots]$ selecting which bindings we want to evaluate or constrain. Both $\triangleright$ and $\triangleleft$ check that $[\hat{y}\ldots]$ and $[\hat{z}{\cdot}l'\ldots]$ have the same number of elements.

When evaluating a location, we simply evaluate *all* the bindings defined for that location on the heap. Thus $\triangleright$ performs the associated measure term as before, but lifts the call to $\triangleright\!\!\triangleright$ with $[\hat{z}{\cdot}l'\ldots]$ as the binding-list. When we return from this call we use $[\hat{y}\ldots]$ to select the required bindings in head normal form.

This is a departure from our previously lazy approach to evaluation. If we wanted to select a subset of the bindings, for example, if we only required the diagonal elements of an array (wherein $[\hat{y}\ldots]$ would contain duplicates), we would be still be evaluating all the bound elements. This may cause the disintegrator to fail more often in cases that evaluate and constrain mutually exclusive sections of an array. Nevertheless, we take the current approach for ease of implementation and assign extensions for future work.

For constraining a location, we require that $[\hat{y}\ldots]$ be a permutation of $[\hat{x}{\cdot}l\ldots]$. In other words, for each invocation of $\triangleleft$ as per $[\hat{x}{\cdot}l\ldots]$, we want to constrain a distinct binding on the heap. This requirement extends to arrays the idea that we cannot observe the same term multiple times. Given this condition, we lift $\lll$ on the associated measure term with $[\hat{x}{\cdot}l\ldots]$. Before we invoke $\lll$ we preserve the order of observation by substituting $[\hat{y}\ldots]$ in place of $[\hat{z}{\cdot}l'\ldots]$ in the measure term.

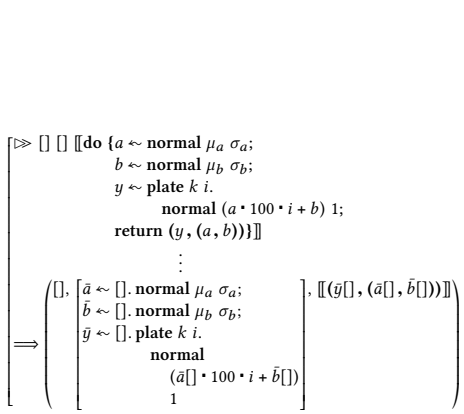
5.3 Putting It All Together

We can now describe a complete method of conditioning the k-measurement **blr** model from eq. (58). The three-step algorithm in eq. (52) is still valid for the new disintegrator:

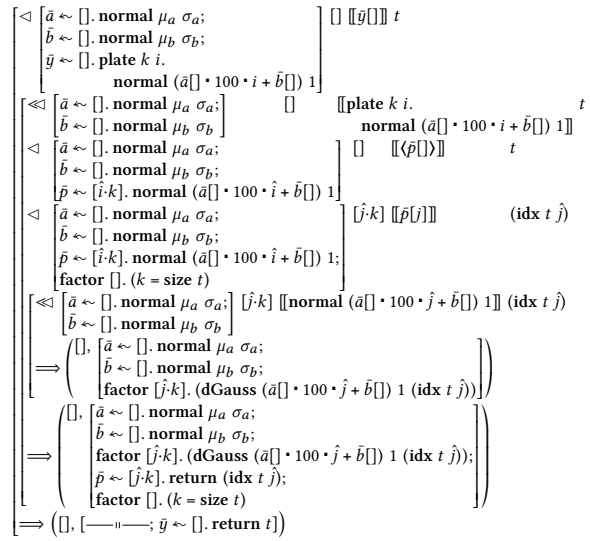
(1) The first step is that of traversing the input term and populating the heap, as shown in fig. 13a.
(2) Fig. 13b shows the second step of performing lifted disintegration on this model. Here we note that there is a constant number of calls to $\triangleleft$. This number is independent of the size of the observed array, in contrast to the case of disintegrating (nested) pairs.
(3) The final step is that of combining the resultant emissions, heap, and residual program:

$$\begin{aligned}
&\textbf{do } \{\bar{a} \leftsquigarrow [].\ \textbf{normal } \mu_a\ \sigma_a; \\
&\quad \bar{b} \leftsquigarrow [].\ \textbf{normal } \mu_b\ \sigma_b; \\
&\quad \textbf{factor } [\hat{j}{\cdot}k].\ (\textbf{dGauss } (\bar{a}[] \cdot 100 \cdot \hat{j} + \bar{b}[])\ 1\ (\textbf{idx } t\ \hat{j})); \\
&\quad \bar{p} \leftsquigarrow [\hat{j}{\cdot}k].\ \textbf{return } (\textbf{idx } t\ \hat{j}); \\
&\quad \textbf{factor } [].\ (k = \textbf{size } t); \\
&\quad \bar{y} \leftsquigarrow [].\ \textbf{return } t; \\
&\quad \textbf{return } (a, b)\}
\end{aligned} \tag{103}$$

All that remains is to convert this program from the internal language to the input language.



(a) Going forward on the input program



(b) Constraining the observed array

Fig. 13. The first two steps of disintegrating k-measurement **blr**

In the case of lifted disintegration, this conversion first involves the following changes:

- Expressions that represent lifted bindings are converted to **plate** expressions:

$$\bar{p} \leftsquigarrow [\hat{x}{\cdot}l{\ldots}].\, m \implies p \leftsquigarrow \mathbf{plate}\ l\ x.\ \ldots m \tag{104}$$

$$\mathbf{factor}\ [\hat{x}{\cdot}l{\ldots}].\, e \implies \_ \leftsquigarrow \mathbf{plate}\ l\ x.\ \ldots \mathbf{do}\ \{\mathbf{factor}\ e; \mathbf{return}\ ()\} \tag{105}$$

- Expressions that select lifted bindings are converted to **idx** expressions:

$$\bar{x}[\hat{x}_1, \ldots, \hat{x}_n] \implies \mathbf{idx}\ (\ldots(\mathbf{idx}\ x\ x_1)\ldots)\ x_n \tag{106}$$

$$\langle \bar{x}[\hat{x}_1, \ldots, \hat{x}_n] \rangle \implies \mathbf{idx}\ (\ldots(\mathbf{idx}\ x\ x_1)\ldots)\ x_n \tag{107}$$

After these changes, the usual beta-reduction and unused-binding-elimination transformations give us the desired posterior representation from eq. (59).

6 EVALUATION

Our disintegrator was evaluated using Hakaru, a suite of program transformations for a language with additional types (such as booleans), operations (such as exp, log, and sin) and distributions (such as binomial, beta, bernoulli, categorical, uniform, and gamma). We looked at the benchmarks distributed with the R2 inference system [Nori et al. 2014], which comprise 11 of the 21 benchmarks used by Gehr et al. [2016] to evaluate their simplifier for probabilistic programs.

Of these 11 benchmarks, 4 models do not use arrays:

- **Burglar alarm**: Probability of burglary given sounding of alarm
- **Grass**: Probability that it is raining given grass is wet
- **Noisy or**: Infer the values of boolean nodes related by noisy-or functions
- **Two coins**: Marginal distribution of two fair coins when at least one landed on tails

Lifted disintegration matches the behavior of Shan and Ramsey's disintegrator in these cases. The machinery for symbolic conditioning of arrays developed in this paper does not hamper the performance of the disintegrator on models that do not observe arrays.

The remaining 7 models use arrays:

- **Clinical trial**: Evaluate a medical treatment based on control and experimental outcomes
- **Coin bias**: Estimate the bias of a coin
- **Digit recognition**: Recognize digits from handwriting
- **HIV**: Tuning the parameters of a linear HIV dynamical model
- **Linear regression**: Fit a line through observed points
- **Survey unbias**: Estimate the gender bias of a Gaussian modeled population
- **True skill**: Use the outcomes of a series of games to infer the skill levels of players

Lifted disintegration correctly handles 6 out of these 7 models. The remaining benchmark—*True skill*—contains conditionals inside arrays, and it is a work in progress to condition such models symbolically and without unrolling arrays.

On these array benchmarks the original disintegration algorithm requires time linear in the number of scalar elements in the observed array. We expect the PSI system of Gehr et al. [2016] to exhibit similar performance, as PSI unrolls loops as well. This is why Gehr et al. evaluate the system on data sets truncated to up to 784 data points.

In contrast, the current approach using lifted disintegration takes time linear in the number of indices used to select an element of the array. All of the benchmarks that we have discussed use either singly-indexed or doubly-indexed arrays, giving lifted disintegration a big performance edge over other symbolic conditioning approaches. In the case of PSI it is tricky to quantify the slowdown because PSI combines conditioning with simplification whereas Hakaru separates these two transformations.

In addition to the R2 benchmarks, lifted disintegration works for various micro-benchmarks that manipulate arrays via mapping, transposing, and copying. The transformation produces posterior distributions in time independent of the number of scalar elements of the array.

7 DISCUSSION

We have extended the automatic disintegrator described by Shan and Ramsey [2017] to condition arrays of stochastic variables. Notably, the extended disintegrator handles arrays *symbolically*, i.e., without unrolling the loops that represent the body of the array. The time taken for our automatic disintegrator to produce the posterior distribution is independent of the length of the array, and linearly proportional to the number of indices needed to select an element. This is only the beginning, as there are many foreseeable extensions to array disintegration.

An important extension would be to handle conditionals. The languages described in this paper lack conditionals since the current system does not handle control-flow uncertainty under arrays. For example, consider the program:

$$
\begin{array}{l}
\textbf{do}\ \{\mu \leftsquigarrow \textbf{normal}\ 0\ 1; \\
\quad p \leftsquigarrow \textbf{plate}\ 50\ \_.\ \textbf{do}\ \{b \leftsquigarrow \textbf{bern}\ 0.5; \\
\qquad\qquad\qquad\qquad \textbf{if}\ b\ (\textbf{normal}\ \mu\ 1)\ (\textbf{normal}\ (\mu + 5)\ 1)\}; \\
\quad \textbf{return}\ (p, \mu)\}
\end{array}
\tag{108}
$$

The **bern** primitive denotes the Bernoulli distribution, which we use here to simulate a fair coin toss. Now the disintegrator lazily flattens the input program, which means that bindings inside loops may end up lifted out into their own array forms. If we pull conditionals out of the body of **plate** in the same manner, we risk changing the meaning of the program. The current system avoids this by failing upon encountering a conditional when disintegration is lifted, i.e., when the current binding-list is non-empty. While we can perceive a solution that duplicates code, we seek a more symbolic solution for a variety of programs that mix arrays and conditionals.

One use-case of conditionals is in expressing array accesses of the form "all elements except one". Such accesses occur in the conditional proposal distributions of Gibbs samplers, which are useful in a wide variety of applications [such as document classification, as described by Resnik and Hardisty 2009]. Extending the current technique to handle conditionals would thus allow us to observe and infer mutually exclusive sections of arrays.

Another significant extension would be to allow dependencies between array elements. The **plate** primitive can only express stochastic loops where elements drawn in one iteration are independent of those in subsequent (or previous) iterations. Looking to the future, we desire symbolic conditioning of arrays where the elements have chain-like dependencies. For example, consider a primitive **chain** having the type:

$$\mathbf{chain} : \mathbb{N} \to s \to (s \to \mathbb{M}\ (\alpha \times s)) \to \mathbb{M}\ (\langle\alpha\rangle \times s) \tag{109}$$

Here the loop body (given by the third argument) is dependent on the previous state (of type s) to produce a measure over some value (of type a) and a new state. Being able to disintegrate programs that use this primitive would be useful for handling Hidden Markov Models and topic models [reviewed by Steyvers and Griffiths 2006].

We believe that the lifted technique described in this paper is an elegant step towards symbolically conditioning arrays. The approach of using indices to represent repetition shows promise in being extensible—we envision using boolean variables to represent conditionals, and generalizing from array-sizes to *shapes* that represent more exotic data structures. We look forward to the role of sophisticated symbolic conditioning in a world of increasingly rich probabilistic models.

ACKNOWLEDGMENTS

Thanks to Jacques Carette, Andre Kuhlenschmidt, Wren Romano, Zachary Sullivan, Robert Zinkov, and anonymous reviewers for helpful comments and discussions.

This research was supported by DARPA grant 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

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2011. Noncomputable Conditional Distributions. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, Washington, DC, USA, 107–116. https://doi.org/10.1109/LICS.2011.49

Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee-Whye Teh. 2009. On Smoothing and Inference for Topic Models. In *UAI*. http://www.ics.uci.edu/~asuncion/pubs/UAI_09.pdf

Patrick Billingsley. 1995. *Probability and Measure*. John Wiley & Sons, New York.

Wray L. Buntine. 1994. Operations for Learning with Graphical Models. *J. Artif. Int. Res.* 2, 1 (Dec. 1994), 159–225. https://doi.org/10.1613/jair.62

Jacques Carette and Chung-Chieh Shan. 2016. *Simplifying Probabilistic Programs Using Computer Algebra*. Springer International Publishing, Cham, 135–152. https://doi.org/10.1007/978-3-319-28228-2_9

George Casella and Edward I. George. 1992. Explaining the Gibbs Sampler. *The American Statistician* 46, 3 (1992), 167–174. http://www.jstor.org/stable/2685208

Joseph T. Chang and David Pollard. 1997. Conditioning as Disintegration. *Statistica Neerlandica* 51, 3 (1997), 287–317.

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. *Bayesian Inference Using Data Flow Analysis*. Technical Report MSR-TR-2013-27. Microsoft Research. http://research.microsoft.com/apps/pubs/default.aspx?id=171611

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely Functional Lazy Nondeterministic Programming. *Journal of Functional Programming* 21, 4–5 (2011), 413–465.

Sebastian Fischer, Josep Silva, Salvador Tamarit, and Germán Vidal. 2008. Preserving Sharing in the Partial Evaluation of Lazy Functional Programs. In *Revised Selected Papers from LOPSTR 2007: 17th International Symposium on Logic-Based Program Synthesis and Transformation (Lecture Notes in Computer Science)*. Springer, Berlin, 74–89.

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. *PSI: Exact Symbolic Inference for Probabilistic Programs*. Springer International Publishing, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A probabilistic programming language for Bayesian inference and optimization. (2015).

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*. http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. Automatic Variational Inference in Stan. *ArXiv e-prints* (June 2015). arXiv:stat.ML/1506.03431

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 144–154.

Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014). http://arxiv.org/abs/1404.0099

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *Neural Information Processing Systems (NIPS)*.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic Models with Unknown Objects. In *Statistical Relational Learning*, Lise Getoor and Ben Taskar (Eds.). MIT Press. http://sites.google.com/site/bmilch/papers/blog-chapter.pdf

Tom Minka, John M. Winn, John P. Guiver, Sam Webster, Yordan Zaykov, Boris Yangel, Alexander. Spengler, and John Bronskill. 2014. Infer.NET 2.6. (2014). http://research.microsoft.com/infernet Microsoft Research Cambridge.

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

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*. AAAI Press, 2476–2482.

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

Philip Resnik and Eric Hardisty. 2009. Gibbs Sampling for the Uninitiated. (2009).

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 130–144. https://doi.org/10.1145/3009837.3009852

Mark Steyvers and Tom Griffiths. 2006. Probabilistic Topic Models. In *Latent Semantic Analysis: A Road to Meaning*., T. Landauer, D. Mcnamara, S. Dennis, and W. Kintsch (Eds.). Laurence Erlbaum. http://cocosci.berkeley.edu/tom/papers/SteyversGriffiths.pdf

Luke Tierney. 1998. A Note on Metropolis-Hastings Kernels for General State Spaces. *The Annals of Applied Probability* 8, 1 (1998), 1–9.

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)*. MIT Press, Cambridge, 770–778.

David Wingate and Theo Weber. 2013. Automated Variational Inference in Probabilistic Programming. *ArXiv e-prints* (Jan. 2013). arXiv:stat.ML/1301.1299

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 1024–1032.