Deriving Probability Density Functions from Probabilistic Functional Programs

Sooraj Bhat[1], Johannes Borgström[2], Andrew D. Gordon[3,4], and Claudio Russo[3]

[1] Georgia Institute of Technology
[2] Uppsala University
[3] Microsoft Research
[4] University of Edinburgh

Abstract. The *probability density function* of a probability distribution is a fundamental concept in probability theory and a key ingredient in various widely used machine learning methods. However, the necessary framework for compiling probabilistic functional programs to density functions has only recently been developed. In this work, we present a density compiler for a probabilistic language with discrete and continuous distributions, and discrete observations, and provide a proof of its soundness. The compiler greatly reduces the development effort of domain experts, which we demonstrate by solving inference problems from various scientific applications, such as modelling the global carbon cycle, using a standard Markov chain Monte Carlo framework.

1 Introduction

Probabilistic programming promises to arm data scientists with declarative languages for specifying their probabilistic models, while leaving the details of how to translate those models to efficient sampling or inference algorithms to a compiler. Many widely used machine learning techniques that might be employed by such a compiler use as input the *probability density function* (PDF) of the model. Such techniques include *maximum likelihood* or *maximum a posteriori estimation*, *L2 estimation*, *importance sampling*, and *Markov chain Monte Carlo* (MCMC) methods.

Despite their utility, density functions have been largely absent from the literature on probabilistic functional programming. This is because the relationship between programs and their density functions is not straightforward: for a given program, the PDF may not exist or may be non-trivial to calculate. Such programs are not merely infrequent pathological curiosities but in fact arise in many ordinary scenarios. In this paper, we define, prove correct, and implement an algorithm for automatically computing PDFs for a large class of programs written in a rich probabilistic programming language.

Probability density functions. We now explain what a probability density function is, where it arises, and what we use it for in this paper. Consider a probabilistic program that generates outcomes from a set Ω. The *probability distribution* $\mathbb{P}$ of the program characterizes its behavior by assigning probabilities to different *subsets (events)* of Ω, denoting the proportion of runs that generate an outcome in that subset.

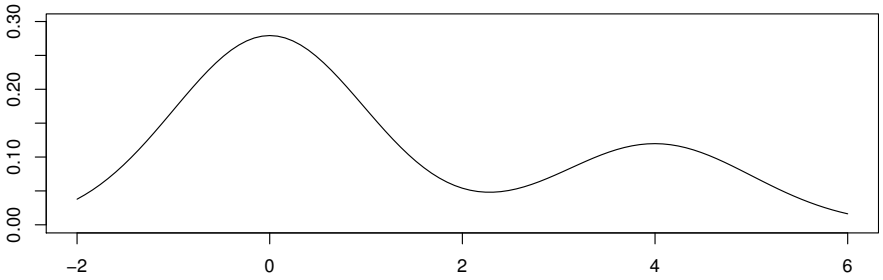
It turns out to be more productive to work with a function on the *elements* of Ω instead of the *subsets* of Ω, which characterizes the distribution. There is always such a function when Ω is countable, known as the *probability mass function*. It is defined $f(x) \triangleq \mathbb{P}(\{x\})$ and enjoys the property $\mathbb{P}(A) = \sum_{x \in A} f(x)$ for all subsets A of Ω. Unfortunately, this construction does not work in the continuous case. Consider a simple *mixture of Gaussians*, here written in Fun (Borgström et al. 2011), a probabilistic functional language embedded within F# (Syme et al. 2007).

```
let w = {mA = 0.0; mB = 4.0} in
  if flip 0.7 then random(Gaussian(w.mA, 1.0)) else random(Gaussian(w.mB, 1.0))
```

This specifies a distribution on the real line (*i.e.* $\Omega = \mathbb{R}$) and corresponds to a generative process where one draws a number from a Gaussian distribution with precision 1.0, and with mean either 0.0 or 4.0 depending on the result of flipping a biased coin. We use a record w with fields mA and mB to hold each mean. Repeating the construction from the discrete case yields the function $g(x) = \mathbb{P}(\{x\})$, which is zero everywhere. Instead we look for a function f such that $\mathbb{P}(A) = \int_A f(x)\,dx$, known as the *probability density function* (PDF) of the distribution. In other words, f is a function where the area under its curve on an interval gives the probability of generating an outcome falling in that interval. The PDF of this program is given by the function

$$f(x) = 0.7 \cdot \mathsf{pdf\_Gaussian}(0.0,\ 1.0,\ \mathsf{x}) + 0.3 \cdot \mathsf{pdf\_Gaussian}(4.0,\ 1.0,\ \mathsf{x})$$

where pdf_Gaussian is the PDF of the Gaussian distribution, the famous "bell curve" from statistics. The function, pictured below, takes higher values where the generative process described above is more likely to generate an outcome.



***Densities functions and* MCMC.** In the example above, the means and variances of the Gaussians, as well as the bias between the two, were known. In this case, the PDF gives a measure of how likely a particular output is. The more common and interesting case in applications is where the parameters are *unknown*, but we have a sample from the process in question. In that case, evaluating the PDF at the sample gives the likelihood of the parameters: a measure of how well a given setting of the parameters matches the sample. We are often interested in properties of the function that maps parameters to their likelihood, *e.g.*, its maximum.

In Bayesian modelling, we use a prior distribution representing our prior beliefs on what the parameters are. Incidentally, this distribution also involves Gaussians, but with a low precision (high variance). To illustrate this, we modify our example as follows:

```
let prior () =
  { mA = random(Gaussian(0.0, 0.001)); mB = random(Gaussian(0.0, 0.001)) }
let moG w =
  if flip 0.7 then random(Gaussian(w.mA, 1.0)) else random(Gaussian(w.mB, 1.0))
let gen w = [| for i in 1 .. 1000 → moG w |]
```

This model generates an array of independent, identically distributed (i.i.d.) points, defined in terms of the single-point model. This lets us capture the idea of seeing many samples generated from the same process.

Markov chain Monte Carlo (MCMC) methods, which generate samples from the posterior distribution, are commonly used for probabilistic inference. The idea of MCMC is to construct a Markov chain in the parameter space of the model, whose equilibrium distribution is the posterior distribution over model parameters. Neal (1993) gives an excellent review of MCMC methods. We here use Filzbach (Purves and Lyutsarev 2012), an adaptive MCMC sampler based on the Metropolis-Hastings algorithm. All that is required for such algorithms is the ability to calculate the posterior density given a set of parameters. The posterior does not need to be from a mathematically convenient family of distributions. Samples from the posterior can then serve as its representation, or be used to calculate marginal distributions of parameters or other integrals under the posterior distribution.

The posterior density is a function of the PDFs of the various pieces of the model, so to perform inference using MCMC, we also need functions to compute the PDFs:

```
let pdf_prior () w = pdf_Gaussian(0.0, 0.001, w.mA) * pdf_Gaussian(0.0, 0.001, w.mB)
let pdf_moG w x = 0.7 * pdf_Gaussian(w.mA, 1.0, x) + 0.3 * pdf_Gaussian(w.mB, 1.0, x)
let pdf_gen w xs = product [| for x in xs → pdf_moG w x |]
```

Filzbach and other MCMC libraries require users to write these three functions, in addition to the probabilistic generative functions prior and gen, which are used for model validation. The goal of this paper is to instead compile these density functions from the generative code. This relieves domain experts from having to write the density code in the first place, as well as from the error-prone task of manually keeping their model code and their density code in synch. Instead, both the PDF and synthetic data are derived from the same declarative specification of the model.

Contributions of this paper. This work defines and applies automated techniques for computing densities to actual inference problems from various scientific applications. The primary technical contribution is a *density compiler* that is correct, useful, and relatively simple and efficient. Specifically:

- We provide the first implementation of a density compiler based on the specification by Bhat et al. (2012). We compile programs in the probabilistic language Infer.NET Fun (described in Section 2) to their corresponding density functions (Section 3).
- We prove that the compilation algorithm is sound (Theorem 1). This is the first such proof for any variant of this compiler.
- We show that the compiler greatly reduces the development effort of domain experts by freeing them from writing densities and that the produced code is comparable in performance to functions hand-coded by experts. We show this on textbook examples and on problems from ecology (Section 4).

2 Fun: Probabilistic Expressions (Review)

We use a version of the core calculus Fun (Borgström et al. 2011) with discrete observations only (implemented using a **fail** construct (Kiselyov and Shan 2009)). Fun is a first-order functional language without recursion that extends the language of Ramsey and Pfeffer (2002), and has a natural semantics in the sub-probability monad. Our implementation efficiently supports a richer language with arrays and array comprehensions, which can be given a semantics in this core.

We use base types **int**, **double** and **unit**, product types (denoting pairs) and sum types (denoting disjoint unions). We let c range over constant data of base type, n over integers and r over real numbers. We write $\mathrm{ty}(c) = t$ to mean that constant c has type t.

Types of Fun:

$$t, u ::= \mathbf{int} \mid \mathbf{double} \mid \mathbf{unit} \mid (t_1 * t_2) \mid (t_1 + t_2)$$

We take $\mathbf{bool} \triangleq \mathbf{unit} + \mathbf{unit}$. We assume a collection of total deterministic functions on these types, including arithmetic and logical operators. For totality, we devine divison by zero to yield zero, i.e. $r/0.0 \triangleq 0.0$. Each operation f of arity n has a signature of the form $\mathbf{val}\ f\colon t_1 * \cdots * t_n \to t_{n+1}$. We also assume standard families of primitive probability distributions of type $\mathsf{PDist}\langle t\rangle$, including the following.

Distributions: $\mathrm{Dist} : (x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle$

$$\begin{aligned}
&\mathsf{Bernoulli} : (\mathsf{bias} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{bool}\rangle\\
&\mathsf{Poisson} : (\mathsf{rate} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{int}\rangle\\
&\mathsf{Gaussian} : (\mathsf{mean} : \mathbf{double} * \mathsf{prec} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle\\
&\mathsf{Beta} : (\mathsf{a} : \mathbf{double} * \mathsf{b} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle\\
&\mathsf{Gamma} : (\mathsf{shape} : \mathbf{double} * \mathsf{scale} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle
\end{aligned}$$

A Bernoulli distribution corresponds to a biased coin flip. The Poisson distribution describes the number of occurrences of independent events that occur at a given average rate. We parameterize the Gaussian distribution by mean and precision. The *standard deviation* σ follows from the identity $\sigma^2 = 1/\mathsf{prec}$. The Beta distribution is a suitable prior for the parameter of Bernoulli distributions; similarly the Gamma distribution is a suitable prior for the parameter of Poisson and the prec parameter of Gaussian.

Expressions of Fun:

$V ::= x \mid c \mid (V,V) \mid \mathbf{inl}_u V \mid \mathbf{inr}_t V$	value
$M, N ::=$	expression
$\quad x \mid c \mid \mathbf{inl}_u\, M \mid \mathbf{inr}_t\, M \mid (M,N)$	value constructors
$\quad \mathbf{fst}\, M \mid \mathbf{snd}\, M$	left/right projection from pair
$\quad f(M)$	primitive operation (deterministic)
$\quad \mathbf{let}\ x = M\ \mathbf{in}\ N$	let (scope of x is N)
$\quad \mathbf{match}\ M\ \mathbf{with}\ \ \mathbf{inl}\ x_1 \to N_1 \mid \mathbf{inr}\ x_2 \to N_2$	matching (scope of x_i is N_i)
$\quad \mathbf{random}(\mathrm{Dist}(M))$	primitive distribution
$\quad \mathbf{fail}_t$	failure

To ensure that a program has at most one type in a given typing environment, **inl** and **inr** are annotated with a type (see (FUN INL) below). The expression **fail** is annotated with the type it is used at. We omit these types where they are not used. When X is a term (possibly with binders), we write $x_1,\ldots,x_n \sharp X$ if none of the x_i appear free in X. We let $\mathrm{op}(M)$ range over $f(M)$, **fst** M, **snd** M, **inl** M and **inr** M; () is the **unit** constant.

We write **observe** M for **if** M **then** () **else fail** and Uniform for Beta(1.0,1.0). When M has sum type, we write **if** M **then** N_1 **else** N_2 for **match** M **with inl** $\_ \to N_1 \mid$ **inr** $\_ \to N_2$.

We write $\Gamma \vdash M : t$ to mean that in type environment $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ (x_i distinct) expression M has type t. Apart from the following, the typing rules are standard. In (FUN INL), (FUN INR) (not shown) and (FUN FAIL), type annotations are used in order to obtain a unique type. In (FUN RANDOM), a random variable drawn from a distribution of type $(x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle$ has type t.

Selected Typing Rules: $\Gamma \vdash M : t$

(FUN INL)
$$\frac{\Gamma \vdash M : t}{\Gamma \vdash \mathbf{inl}_u\ M : t + u}$$

(FUN FAIL)
$$\frac{}{\Gamma \vdash \mathbf{fail}_t : t}$$

(FUN RANDOM)
$$\frac{\mathrm{Dist} : (x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle \qquad \Gamma \vdash M : (t_1 * \cdots * t_n)}{\Gamma \vdash \mathbf{random}(\mathrm{Dist}(M)) : t}$$

Semantics As usual, for precision concerning probabilities over uncountable sets, we turn to measure theory. The interpretation of a type t is the set $\mathbf{V}_t$ of closed values of type t (real numbers, integers etc.). Below we consider only Lebesgue-measurable sets of values, defined using the standard (Euclidian) metric, and ranged over by A, B.

A measure μ over t is a function, from (measurable) subsets of $\mathbf{V}_t$ to the non-negative real numbers extended with ∞, that is σ-additive, that is, $\mu(\varnothing) = 0.0$ and $\mu(\cup_i A_i) = \Sigma_i \mu(A_i)$ if $A_1, A_2, \ldots$ are pair-wise disjoint. The measure μ is called a probability measure if $\mu(\mathbf{V}_t) = 1.0$, and a sub-probability measure if $\mu(\mathbf{V}_t) \leq 1.0$.

We associate a default or *stock* measure to each type, inductively defined as the counting measure on $\mathbb{Z}$ and $\{()\}$, the Lebesgue measure on $\mathbb{R}$, and the Lebesgue-completion of the product and disjoint sum, respectively, of the two measures for $t * u$ and $t + u$. If f is a non-negative (measurable) function $t \to$ **double**, we let $\int f$ be the Lebesgue integral of f with respect to the stock measure on t, if the integral is defined. This integral coincides with $\Sigma_{x \in \mathbf{V}_t} f(x)$ for discrete types t, and with the standard Riemann integral (if it is defined) on $t =$ **double**. We write $\int f(x)\, dx$ for $\int \lambda x.f(x)$, and $\int f(x)\, d\mu(x)$ for Lebesgue integration with respect to the measure μ on t. The Iverson brackets $[p]$ are 1.0 if predicate p is true, and 0.0 otherwise. We write $\int_A f$ for $\int \lambda x.[x \in A] \cdot f(x)$. Let g be a *density* of μ (with respect to the stock measure) if $\int_A 1\, d\mu(x) = \int_A g$ for all A. If μ is a (sub-)probability measure, then we say that g as above is its PDF.

The semantics of a closed Fun expression M is a sub-probability measure $\mathbb{P}_M$ over its return type. Open **fail**-free Fun expressions have a straightforward semantics (Ramsey and Pfeffer 2002) in the probability monad (Giry 1982). In order to treat the **fail** primitive, our extension (Gordon et al. 2013) of Ramsey and Pfeffer's semantics uses

a richer monad: the sub-probability monad (Panangaden 1999)[5]. In the sub-probability monad, bind and return are defined in the same way as the probability monad; it is only the set of admissible measures μ that is extended to admit $|\mu| \leq 1$. The semantics of an expression M is a sub-probability measure. Below, σ is a substitution, that gives values to the free variables of M.

Monadic Semantics of Fun with fail, $\mathscr{P}[\![M]\!]\ \sigma$: assuming $z \sharp N, N_1, x, x_1, x_2, \sigma$

$(\mu \gg\!= f)\ A \triangleq \int f(x)(A)\, d\mu(x)$	Monadic bind
$(\texttt{return}\, V)\ A \triangleq 1$ if $V \in A$, else 0	Monadic return
$\texttt{zero}\ A \triangleq 0$	Monadic zero

$\mathscr{P}[\![x]\!]\ \sigma \triangleq \texttt{return}\ (x\sigma)$
$\mathscr{P}[\![c]\!]\ \sigma \triangleq \texttt{return}\, c$
$\mathscr{P}[\![\mathrm{op}(M)]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg\!= \texttt{return} \circ \mathrm{op}$
$\mathscr{P}[\![(M,N)]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg\!= \lambda z.\mathscr{P}[\![N]\!]\ \sigma \gg\!= \lambda w.\texttt{return}\ (z,w)$
$\mathscr{P}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg\!= \lambda z.\mathscr{P}[\![N]\!]\ (\sigma, x \mapsto z)$

$\mathscr{P}[\![\textbf{match}\ M\ \textbf{with inl}\ x_1 \to\ N_1 \mid \textbf{inr}\ x_2 \to\ N_2]\!]\ \sigma \triangleq$
$\quad \mathscr{P}[\![M]\!]\ \sigma \gg\!= \texttt{either}\ (\lambda z.\mathscr{P}[\![N_1]\!]\ (\sigma, x_1 \mapsto z))\ (\lambda z.\mathscr{P}[\![N_2]\!]\ (\sigma, x_2 \mapsto z))$

$\mathscr{P}[\![\textbf{random}(\mathrm{Dist}(M))]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg\!= \lambda z.\mu_{\mathrm{Dist}(z)}$
$\mathscr{P}[\![\textbf{fail}]\!]\ \sigma \triangleq \texttt{zero}$

Here $\texttt{either}\ f\ g\ (\textbf{inl}\, V) \triangleq f\, V$ and $\texttt{either}\ f\ g\ (\textbf{inr}\, V) \triangleq g\, V$. We let the semantics of a closed expression M be $\mathbb{P}_M \triangleq \mathscr{P}[\![M]\!]\ \varepsilon$, where ε denotes the empty substitution.

3 The Density Compiler

We compute the PDF of a Fun program by compilation into a deterministic language, that features integration as a primitive operation. In our implementation, we call out to a numeric integration library to compute the value of integrals. Our compilation is based on that of Bhat et al. (2012), with modifications to treat **fail** statements, deterministic let bindings, **match** (and general **if**) statements, and integer arithmetic.

3.1 Target Language for Density Computations

For our target language, we choose a standard deterministic functional language, augmented with stock integration.

Expressions of the Target Language: E, F

$T, U ::= \textbf{int} \mid \textbf{double} \mid \textbf{unit} \mid T \to U \mid T + U \mid T * U$	target types
$E, F ::=$	target expression
$\quad x \mid c \mid \textbf{inl}_U\ E \mid \textbf{inr}_T\ E \mid (E, F)$	value constructors

[5] Sub-probabilities are also useful to reason about our compilation of **match** (and **if**) statements, where the probability that we have entered a particular branch may be less than 1.

$\mathbf{fst}\ E \mid \mathbf{snd}\ E \mid f(E)$	deterministic operations
$\mathbf{let}\ x = E\ \mathbf{in}\ F$	let (scope of x is F)
$\mathbf{match}\ E\ \mathbf{with}\ \ \mathbf{inl}\ x_1 \to F_1 \mid \mathbf{inr}\ x_2 \to F_2$	matching (scope of x_i is F_i)
$\lambda(x_1,...,x_n).\ E$	lambda abstraction
$E\ F$	application
$\int E$	stock integration
$\bot_T$	failure

The typing rules for integration and failure are as follows (the other typing rules are standard):

Selected Typing Rules: $\Gamma \vdash E : T$

(TARGET INT)

$$\frac{\Gamma \vdash E : T \to \mathbf{double} \qquad T \text{ is a first-order type}}{\Gamma \vdash \int E : \mathbf{double}}$$

(TARGET FAIL)

$$\frac{}{\Gamma \vdash \bot_T : T}$$

Small-step CBV-evaluation $\to$ of well-typed expressions is standard, except for short-circuiting multiplication: $0.0 \cdot E \to 0.0$, avoiding failures in E. Evaluation can fail either explicitly ($\bot$) or by evaluating an undefined integral, *e.g.* $\int \lambda x.\sin x \to \bot_{\mathbf{double}}$.

3.2 Relational Specification of the Compiler

The translation is based on the let-structure of the expression. Variables that are let-bound in outer lets are referred to as parameters, and a context gathers random and deterministic inner lets.

Probability Context:

$\Upsilon ::=$	probability context
ε	empty context
Υ, x	random variable
$\Upsilon, x = E$	deterministic variable

A probabilistic context Υ is often used together with a density expression (E below), which is an open term that expresses the joint probability density of the random variables in the context and the constraints that have been collected when choosing branches in **match** statements. The main judgment is $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$, which computes a function F from return values of M to densities, where parameters may occur free in F. The marginal judgment $\Upsilon; E \vdash \text{marg}(x_1, \ldots, x_k) \Rightarrow F$ yields the joint PDF of its argument, marginalizing out all other random variables in Υ.

Inductively Defined Judgments of the Compiler:

$\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$	in $\Upsilon; E$ expression F gives the PDF of M
$\Upsilon; E \vdash \text{marg}(x_1, \ldots, x_k) \Rightarrow F$	in $\Upsilon; E$ expression F gives the PDF of $(x_1, \ldots, x_k)$

For a probability context to be well-formed, it has to be well-scoped and well-typed.

Well-formed probability context: $\Gamma \vdash \Upsilon$ wf

$$\frac{}{\Gamma \vdash \varepsilon \text{ wf}}\ (\textsc{Env Empty}) \qquad \frac{\Gamma \vdash \Upsilon \text{ wf} \qquad \Gamma \vdash x : t \qquad x \sharp \Upsilon}{\Gamma \vdash \Upsilon, x \text{ wf}}\ (\textsc{Env Var}) \qquad \frac{\Gamma \vdash \Upsilon \text{ wf} \qquad \Gamma \vdash x : t \qquad x \sharp \Upsilon \qquad \Gamma \vdash E : t}{\Gamma \vdash \Upsilon, x = E \text{ wf}}\ (\textsc{Env Const})$$

Given a well-formed context Υ, we can extract the random variables $\text{rands}(\Upsilon)$, and an idempotent substitution σ_Υ that describes the deterministic variables.

Random variables $\text{rands}(\Upsilon)$ **and values of deterministic variables** σ_Υ

$$\begin{aligned} \text{rands}(\varepsilon) &\triangleq \varepsilon & \sigma_\varepsilon &\triangleq [] \\ \text{rands}(\Upsilon, x) &\triangleq \text{rands}(\Upsilon), x & \sigma_{\Upsilon, x} &\triangleq \sigma_\Upsilon \\ \text{rands}(\Upsilon, x = E) &\triangleq \text{rands}(\Upsilon) & \sigma_{\Upsilon, x=E} &\triangleq [x \mapsto E\sigma_\Upsilon]\sigma_\Upsilon \end{aligned}$$

We define "M det" to hold iff M does not contain any occurrence of **random** or **fail**. If M det holds, then M is also an expression in the target language syntax, and we silently treat it as such (in rules (LET DET) and (MATCH DET), for example). If M det and $\text{rands}(\Upsilon) \sharp (M\sigma_\Upsilon)$, then M is constant under Υ.

The marg judgment yields the joint marginal PDF of the random variables in its argument. To compute the PDF, we first substitute in the deterministic **let**-bound variables, and then integrate out the remaining random variables. Except for rule (DISCRETE) below, $\text{marg}(x_1, ..., x_k)$ is used with $k \in \{0, 1, 2\}$; the case $k = 0$ is used to compute the probability of being in the current branch of the program.

Marginal Density: $\Upsilon; E \vdash \text{marg}(x_1, ..., x_k) \Rightarrow F$

$$\frac{\{x_1, ..., x_k\} \cup \{y_1, ..., y_n\} = \text{rands}(\Upsilon) \qquad x_1, ..., x_k, y_1, ..., y_n \text{ distinct}}{\Upsilon; E \vdash \text{marg}(x_1, ..., x_k) \Rightarrow \lambda(x_1, ..., x_k).\ \int \lambda(y_1, ..., y_n).\ E\sigma_\Upsilon}\ (\textsc{Marginal})$$

The dens judgment gives the density F of M in the current context Υ, where E is the accumulated body of the density function so far. We introduce fresh lambda-bound variables in the result F; below we assume that $z, w \sharp \Upsilon, E, M$.

Density Compiler, base cases: $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$

$$\frac{(x = E') \in \Upsilon \qquad \Upsilon; E \vdash \text{dens}(E') \Rightarrow F}{\Upsilon; E \vdash \text{dens}(x) \Rightarrow F}\ (\textsc{Var Det}) \qquad \frac{x \in \text{rands}(\Upsilon) \qquad \Upsilon; E \vdash \text{marg}(x) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(x) \Rightarrow F}\ (\textsc{Var Rnd})$$

$$\frac{\text{ty}(c) \text{ countable} \qquad \Upsilon; E \vdash \text{marg}(\varepsilon) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(c) \Rightarrow \lambda z.\ [z = c] \cdot (F\ ())}\ (\textsc{Constant}) \qquad \frac{}{\Upsilon; E \vdash \text{dens}(\textbf{fail}) \Rightarrow \lambda z.\ 0.0}\ (\textsc{Fail})$$

For a deterministic variable, (VAR DET) recurses into its definition. The rule (VAR RND) computes the marginal density of a random variable using the marg judgment. The (CONSTANT) rule states that the probability density of a discrete constant c (built from sums and products of integers and units) is the probability of being in the current branch at c, and 0 elsewhere. The (FAIL) rule gives that the density of **fail** is zero.

Density Compiler, sums and tuples: $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$

$$\frac{\Upsilon; E \vdash \text{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(\textbf{inl}\ M) \Rightarrow \mathsf{either}\ F\ (\lambda\_.\, 0)}\ (\textsc{Sum Con L}) \qquad \frac{\Upsilon; E \vdash \text{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(\mathsf{fromL}(M)) \Rightarrow \lambda z.(F\ (\textbf{inl}\ z))}\ (\textsc{fromL})$$

$$\frac{\Upsilon; E \vdash \text{marg}(x,y) \Rightarrow F}{\Upsilon; E \vdash \text{dens}((x,y)) \Rightarrow F}\ (\textsc{Tuple Var}) \qquad \frac{\Upsilon; E \vdash \text{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(\textbf{fst}\ M) \Rightarrow \lambda z.\ \int \lambda w.\ F\ (z,w)}\ (\textsc{Tuple Proj L})$$

Symmetric versions of (SUM CON L), (TUPLE PROJ L) and (FROML) are omitted above. (SUM CON L) states that the density of **inl** M is the density of M in the left branch of a sum, and 0 in the right. Its dual is (FROML). The rule (TUPLE VAR) computes the joint marginal density of two random variables. (This syntactic restriction can be lifted by considering dependency information for the expressions in the tuple (Bhat et al. 2012).) (TUPLE PROJ L) marginalizes out the left dimension of a pair.

Density Compiler, let and match: $\Upsilon; E \vdash \text{dens}(\textbf{let}\ x = M\ \textbf{in}\ N) \Rightarrow F$

$$\frac{M\ \text{det} \qquad \Upsilon, x = M; E \vdash \text{dens}(N) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(\textbf{let}\ x = M\ \textbf{in}\ N) \Rightarrow F}\ (\textsc{Let Det}) \qquad \frac{\neg(M\ \text{det}) \qquad \varepsilon; 1 \vdash \text{dens}(M) \Rightarrow F_1 \qquad \Upsilon, x; E \cdot (F_1\ x) \vdash \text{dens}(N) \Rightarrow F_2}{\Upsilon; E \vdash \text{dens}(\textbf{let}\ x = M\ \textbf{in}\ N) \Rightarrow F_2}\ (\textsc{Let Rnd})$$

The rule (LET DET) simply adds a deterministic let-binding to the context. In (LET RND), we compute the density of the let-bound variable in an empty context, and multiply it into the current accumulated density when computing the density of the body.

Below, we let $\mathsf{isL} := \lambda x.\textbf{if}\ x\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0$ be the indicator function for the left branch, and dually for isR. We also use a deterministic operation $\mathsf{fromL} : t + u \to t$ such that $\mathsf{fromL}(M) \to \textbf{match}\ M\ \textbf{with inl}\ x \to\ x \mid \textbf{inr}\ y \to\ \bot_t$, and its dual fromR.

Density Compiler, rules for match: $\Upsilon; E \vdash \text{dens}(\textbf{match}\ M\ \textbf{with} \ldots) \Rightarrow F$

$$\frac{M\ \text{det} \qquad \begin{array}{l}\Upsilon, y_1 = \mathsf{fromL}(M); E \cdot (\mathsf{isL}\ M\sigma_\Upsilon) \vdash \text{dens}(N_1) \Rightarrow F_1 \\ \Upsilon, y_2 = \mathsf{fromR}(M); E \cdot (\mathsf{isR}\ M\sigma_\Upsilon) \vdash \text{dens}(N_2) \Rightarrow F_2\end{array}}{\Upsilon; E \vdash \text{dens}(\textbf{match}\ M\ \textbf{with inl}\ y_1 \to\ N_1 \mid \textbf{inr}\ y_2 \to\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}\ (\textsc{Match Det})$$

$$\frac{\begin{array}{rl}\neg(M\ \text{det}) & \Upsilon, y_1; E \cdot (F\ (\textbf{inl}\ y_1)) \vdash \text{dens}(N_1) \Rightarrow F_1 \\ \varepsilon; 1 \vdash \text{dens}(M) \Rightarrow F & \Upsilon, y_2; E \cdot (F\ (\textbf{inr}\ y_2)) \vdash \text{dens}(N_2) \Rightarrow F_2\end{array}}{\Upsilon; E \vdash \text{dens}(\textbf{match}\ M\ \textbf{with inl}\ y_1 \to\ N_1 \mid \textbf{inr}\ y_2 \to\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}\ (\textsc{Match Rnd})$$

(MATCH DET) is based on (LET DET), and we multiply the constraint that we are in the correct branch ($\mathsf{isL}\, M\sigma_\Upsilon$ or $\mathsf{isR}\, M\sigma_\Upsilon$) with the joint density expression. We also employ deterministic functions fromL and fromR to avoid recursive calls to (MATCH DET) when computing the density of the match-bound variable. The (MATCH RND) rule is based on (LET RND), and we again multiply in the constraint that we are in the left (or right) branch of the **match**.

Density Compiler, random variables : $\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F$

(RANDOM CONST)

$$\frac{M \text{ det} \quad \mathrm{rands}(\Upsilon) \sharp (M\sigma_\Upsilon) \quad \Upsilon; E \vdash \mathrm{marg}(\varepsilon) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{random}(\mathrm{Dist}(M))) \Rightarrow \lambda z.\ (\mathrm{pdf}_{\mathrm{Dist}(M\sigma_\Upsilon)}\ z) \cdot (F\ ())}$$

(RANDOM RND)

$$\frac{\neg(M \text{ det} \wedge \mathrm{rands}(\Upsilon) \sharp (M\sigma_\Upsilon)) \quad \Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{random}(\mathrm{Dist}(M))) \Rightarrow \lambda z. \int \lambda w. (\mathrm{pdf}_{\mathrm{Dist}(w)}\ z) \cdot (F\ w)}$$

In (RANDOM CONST), a random variable drawn from a primitive distribution with a constant argument has the expected PDF (multiplied with the probability that we are in the current branch). (RANDOM RND) treats calls to **random** with a random argument by marginalizing over the argument to the distribution.

In **if** statements, the branching expression is of type $\mathbf{bool} = \mathbf{unit} + \mathbf{unit}$, so we can make a straightforward case distinction.

Derived rule for if statements

(IF DET)

$$\frac{M \text{ det} \quad \Upsilon; E \cdot [M\sigma_\Upsilon = \mathbf{true}] \vdash \mathrm{dens}(N_1) \Rightarrow F_1 \quad \Upsilon; E \cdot [M\sigma_\Upsilon = \mathbf{false}] \vdash \mathrm{dens}(N_2) \Rightarrow F_2}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}$$

For numeric operations on real numbers we mimic the change of variable rule of integration (often summarized as "$dx = \frac{dx}{dy} dy$"), multiplying the density of the argument with the derivative of the inverse operation. This is exemplified by the following rules.

Density compiler, numeric operations on reals : $\Upsilon; E \vdash \mathrm{dens}(f(M)) \Rightarrow F$

(NEG)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(-M) \Rightarrow \lambda z.\ F\ (-z)}$$

(INVERSE)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(1/M) \Rightarrow \lambda z.\ (F\ 1/z) \cdot (1/z^2)}$$

(EXP)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathsf{exp}(M)) \Rightarrow \lambda z.\ \mathbf{if}\ z > 0.0\ \mathbf{then} (F\ \mathsf{log}(z)) \cdot (1/z)\ \mathbf{else}\ 0.0}$$

(TRANSLATE)

$$\frac{N \text{ det} \quad \mathrm{rands}(\Upsilon) \sharp (N\sigma_\Upsilon) \quad \Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(M+N) \Rightarrow \lambda z.\ F\ (z - N\sigma_\Upsilon)}$$

(PLUS)

$$\frac{\Upsilon; E \vdash \text{dens}((M,N)) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(M+N) \Rightarrow \lambda z.\ \int \lambda w.\ F\ (w, z-w)}$$

The (DISCRETE) rule for discrete operations such as logical and comparison operations and integer arithmetic computes the expectation of an indicator function over the joint distribution of the random variables occurring in the expression.

Density compiler, discrete operations : $\Upsilon; E \vdash \text{dens}(f(M)) \Rightarrow F$

(DISCRETE)

$$\frac{f : t \to u \quad u \text{ discrete} \quad M \text{ det} \quad \bar{y} = \text{rands}(\Upsilon) \cap \text{fv}(M\sigma_\Upsilon) \quad \Upsilon; E \vdash \text{marg}(\bar{y}) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(f(M)) \Rightarrow \lambda z.\ \int \lambda \bar{y}.\ [z = f(M\sigma_\Upsilon)] \cdot (F\ \bar{y})}$$

These derived judgments relate the types of the various terms occurring in the marg and dens judgments.

Lemma 1 (Derived Judgments).
If $\Gamma, \Gamma_\Upsilon \vdash \Upsilon$ wf *and* $\text{dom}(\Gamma_\Upsilon) = \text{rands}(\Upsilon) \cup \text{dom}(\sigma_\Upsilon)$ *and* $\Gamma, \Gamma_\Upsilon \vdash E : \textbf{double}$ *then*

(1) *If* $\Upsilon; E \vdash \text{marg}(x_1, \ldots, x_n) \Rightarrow F$ *and* $\Gamma_\Upsilon \vdash (x_1, \ldots, x_n) : (t_1 * \cdots * t_n)$ *then* $\Gamma \vdash F : (t_1 * \cdots * t_n) \to \textbf{double}$.
(2) *If* $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$ *and* $\Gamma, \Gamma_\Upsilon \vdash M : t$ *then* $\Gamma \vdash F : t \to \textbf{double}$.

The soundness theorem asserts that, for all closed expressions M, the density function given by the density compiler indeed characterizes (via stock integration) the distribution of M given by the monadic semantics:

Theorem 1 (Soundness). *If* $\varepsilon; 1 \vdash \text{dens}(M) \Rightarrow F$ *and* $\varepsilon \vdash M : t$ *then*

$$(\mathscr{P}[\![M]\!]\ \varepsilon)\ A = \int_A F$$

Proof: By joint induction on the derivations of $\text{dens}(M)$ and $M : t$, using the following induction hypothesis: if $\Gamma, \Gamma_\Upsilon \vdash \Upsilon$ wf and $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$ and $\Gamma, \Gamma_\Upsilon \vdash M : t$ and $\Gamma, \Gamma_\Upsilon \vdash E : \textbf{double}$ and $\Gamma \vdash \rho$ and $\text{dom}(\Gamma_\Upsilon) = \text{rands}(\Upsilon) \cup \text{dom}(\sigma_\Upsilon)$ and $|\mu| \leq 1$ and $\mu(B) = \int_B \lambda(\text{rands}(\Upsilon)).E\rho$, and $(\forall x \in \text{dom}(\sigma_\Upsilon) \forall \rho'.\ \Gamma_\Upsilon \vdash \rho'$ and $\sigma_\Upsilon(x)\rho\rho' \to^* \bot$ implies that $E\rho\rho' \to^* 0.0)$ then

$$(\mu \gg\!= (\lambda(\text{rands}(\Upsilon)).(\mathscr{P}[\![M]\!]\ (\sigma_\Upsilon\rho))))\ A = \int_A F\rho$$

where $\Gamma \vdash \rho$ is defined as $\varepsilon \vdash \varepsilon$, and $\Gamma, x : t \vdash \rho[x \mapsto V]$ when $\varepsilon \vdash V : t$ and $\Gamma \vdash \rho$. ■

The induction hypothesis on evaluation of $\sigma_\Upsilon(x)\rho\rho'$ above is used when attempting to evaluate match-bound variables for valuations that give the other branch. For such valuations the density becomes zero, because of the short-circuiting property of multiplication by 0.0.

As an example of compilation, the **if** statement in the program

let p = **random**(Uniform) **in let** b = **random**(Bernoulli(p)) **in if** b **then** p+1.0 **else** p

is handled by (IF DET), yielding a density function that is β-equivalent to

$$\lambda z. \quad \int \lambda b.[0 \le z-1 \le 1] \cdot (\textbf{if } b \textbf{ then } z-1 \textbf{ else } 2-z) \cdot [b = \textbf{true}] \\ + \int \lambda b.[0 \le z \le 1] \cdot (\textbf{if } b \textbf{ then } z \textbf{ else } 1-z) \cdot [b = \textbf{false}]$$

which simplifies to the (V-shaped) function $\lambda z.[1 \le z \le 2] \cdot (z-1) + [0 \le z \le 1] \cdot (1-z)$.

4 Evaluation

We evaluate the compiler on several synthetic textbook examples and several real examples from scientific applications. We wish to validate that the density compiler handles these examples, and understand how much the compiler reduces the developer burden, and its performance impact.

Implementation. Since Fun is a sublanguage of F#, we implement our models as F# programs, and use the quotation mechanism of F# to capture their syntax trees. Running the F# program corresponds to sampling data from the model. To compute the PDF, the compiler takes the syntax tree (of F# type Expr) of the model and produces another Expr corresponding to a deterministic F# program as output. We then use run-time code generation to compile the generated Expr to MSIL bytecode, which is just-in-time compiled to executable machine code when called, just as for statically compiled F# code. Our implementation supports arrays and records, which are both translated using adaptations of the corresponding rules for tuples. For efficiency, the implementation must avoid introducing redundant computations, translating the use of substitution in the formal rules to more efficient **let**-bindings that share the values of expressions that would otherwise be re-computed. As is common practice, our implemenation and Filzbach both work with the *logarithm* of the density, which avoids products of densities in favor of sums of log-densities where possible, to avoid numerical underflow.

Metrics. We consider scientific models with existing implementations for MCMC-based inference, written by domain experts. We are interested in how the modelling and inference experience would change, in terms of developer effort and performance impact, when adopting the Fun-based solution.

We assess the reduction in developer burden by measuring the code sizes (in lines-of-code (LOC)) of the original implementations of model and density code, and of the corresponding Fun model. For the synthetic examples, we have written both the model and the density code. The original implementations of the scientific models contain helper code such as I/O code for reading and writing data files in an application-specific format. Our LOC counts do not consider such helper code, but only count the code for generating synthetic data from the model, code for computing the logarithm of the posterior density of the model, and model-related code for setting up and interacting

Example	orig	LOC, orig	LOC, Fun		time (s), orig	time (s), Fun	
mixture of Gaussians	F#	32	20	0.63x	1.77	4.78	2.7x
linear regression	F#	27	18	0.67x	0.63	2.08	3.3x
species distribution	C#	173	37	0.21x	79	189	2.4x
net primary productivity	C#	82	39	0.48x	11	23	2.1x
global carbon cycle	C#	1532	402	0.26x	n/a	764	n/a

Table 1. Lines-of-code and running time comparisons of synthetic and scientific models.

with Filzbach itself. We also compare the running times of the original implementations versus the Fun versions for MCMC-based inference using Filzbach, not including data manipulation before and after running inference.

4.1 Examples

Synthetic examples. Our synthetic examples are models for two classic problems in statistics and machine learning: the supervised learning task *linear regression*, and the unsupervised learning task *mixture of Gaussians*. The latter can be thought of as a probabilistic version of *k-means clustering*. In linear regression, inference is trying to determine the coefficients of the line. In mixture of Gaussians, inference is trying to determine the unknown mixing bias and the means and variances of the Gaussian components.

Species distribution. The species distribution problem is to give the probability that certain species will be present at a given site, based on climate factors. It is a problem of long-standing interest in ecology and has taken on new relevance in light of the issue of climate change. The particular model that we consider is designed to mitigate *regression dilution* arising from uncertainty in the predictor variables, for example, measurement error in temperature data (McInerny and Purves 2011). Inference tries to determine various features of the species and the environment, such as the optimal temperature preferred by a species, or the true temperature at a site.

Global carbon cycle. The dynamics of the Earth's climate are intertwined with the terrestrial carbon cycle, and better carbon models (modelling how carbon in the air gets converted to biomass) enable better constrained projections about these systems. We consider a fully data-constrained terrestrial carbon model by Smith et al. (2012). It is a composition of various submodels for smaller processes such as *net primary productivity*, the fine root mortality rate or the fraction of trees that are evergreen versus deciduous. Inference tries to determine the different parameters of these submodels.

Discussion. Table 1 reports the metrics for each example. The LOC numbers show significant reduction in code size, with more significant savings as the size of the model grows. The larger models (where the Fun versions are $\approx 25\%$ of the size of the original) are more indicative of the savings in developer and maintenance effort, since smaller

models have a larger fraction of boiler-plate code. We find the running times encouraging: we have made little attempt to optimize the generated code, and preliminary testing indicates that much of the performance slow-down is due to constant factors.

The global carbon cycle model is composed of submodels, each with their own dataset. Unfortunately, it is unclear from the original source code how this composition translates to a run of inference, making it difficult to know what constitutes a fair comparison. Thus, we do not report a running time for the full model. However, we can measure the running time of individual submodels, such as net primary productivity, where the data and control flow are simpler.

5 Related Work

The most closely related work to this paper is recent work by Bhat et al. (2012) who develop a theoretical framework for computing PDFs, but describe no implementation nor correctness proof. The density compiler of Section 3 has a simpler presentation, with two judgments compared to five, and has rules for deterministic **let**s and operations on integers. Our paper also uses a richer language (Fun), which adds **fail**, **match** and general **if** (and for performance reasons, deterministic **let**).

Gordon et al. (2013) describe a naive density calculation routine for Fun without random **let**s; this sublanguage does not cover many useful classes of models such as hierarchical and mixture models.

The BUGS system computes densities from declaratively specified models to perform Gibbs sampling (Gilks et al. 1994). However, the models are not compositional as in this work, and only the joint density over all variables is possible. The AutoBayes system also computes densities for deriving maximum likelihood and Bayesian estimators for a significant class of statistical models (Schumann et al. 2008). It is not formally specified and does not appear to be compositional. Neither system addresses the nonexistence of PDFs, presumably restricting expressivity in order to avoid the issue.

Inference for the Church language also uses MCMC, but works with distributions over the runs of a program instead of over its return value (Wingate et al. 2011).

6 Conclusions and Future Work

We have described a compiler for automatically computing probability density functions for programs from a rich Bayesian probabilistic programming language, proven the algorithm correct, and shown its applicability to real-world scientific models.

The inclusion of **fail** in the language appears highly useful for scientific models, giving a simple facility to exclude branches that are scientifically impossible from consideration. However, more investigation is needed to settle this claim.

Techniques from automatic differentiation (Griewank and Walther 2008) may be useful to treat higher-dimensional primitive probability distributions.

A drawback of the compiler is that terms of composite type are required either to have a PDF or to be deterministic, ruling out terms such as (0.0, **random**(Uniform)). One possibility for future work would be to refine the types of expressions with determinacy information, and make use of this additional information in the compiler.

Bibliography

S. Bhat, A. Agarwal, R. W. Vuduc, and A. G. Gray. A type theory for probability density functions. In J. Field and M. Hicks, editors, *POPL*, pages 545–556. ACM, 2012.

J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 77–96. Springer, 2011. Download available at `http://research.microsoft.com/fun`.

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43:169–178, 1994.

M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin / Heidelberg, 1982.

A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. Nori, S. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *POPL*, 2013.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

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

G. McInerny and D. Purves. Fine-scale environmental variation in species distribution modelling: regression dilution, latent variables and neighbourly advice. *Methods in Ecology and Evolution*, 2(3):248–257, 2011.

R. M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto, September 1993.

P. Panangaden. The category of Markov kernels. *Electronic Notes in Theoretical Computer Science*, 22:171–187, 1999.

D. Purves and V. Lyutsarev. *Filzbach User Guide*, 2012. Available at `http://research.microsoft.com/en-us/um/cambridge/groups/science/tools/filzbach/filzbach.htm`.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

J. Schumann, T. Pressburger, E. Denney, W. Buntine, and B. Fischer. AutoBayes program synthesis system users manual. Technical Report NASA/TM–2008–215366, NASA Ames Research Center, 2008.

M. J. Smith, M. C. Vanderwel, V. Lyutsarev, S. Emmott, and D. W. Purves. The climate dependence of the terrestrial carbon cycle; including parameter and structural uncertainties. *Biogeosciences Discussions*, 9:13439–13496, 2012.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th Intl. Conf. on Artificial Intelligence and Statistics*, page 131, 2011.