A Model-Learner Pattern for Bayesian Reasoning

Andrew D. Gordon (Microsoft Research and University of Edinburgh) Mihhail Aizatulin (Open University)
Johannes Borgström (Uppsala University) Guillaume Claret (Microsoft Research)
Thore Graepel (Microsoft Research) Aditya V. Nori (Microsoft Research)
Sriram K. Rajamani (Microsoft Research) Claudio Russo (Microsoft Research)

Abstract

A Bayesian model is based on a pair of probability distributions, known as the *prior* and *sampling* distributions. A wide range of fundamental machine learning tasks, including regression, classification, clustering, and many others, can all be seen as Bayesian models. We propose a new probabilistic programming abstraction, a *typed Bayesian model*, based on a pair of probabilistic expressions for the prior and sampling distributions. A *sampler* for a model is an algorithm to compute synthetic data from its sampling distribution, while a *learner* for a model is an algorithm for probabilistic inference on the model. Models, samplers, and learners form a generic programming pattern for model-based inference. They support the uniform expression of common tasks including model testing, and generic compositions such as mixture models, evidence-based model averaging, and mixtures of experts. A formal semantics supports reasoning about model equivalence and implementation correctness. By developing a series of examples and three learner implementations based on exact inference, factor graphs, and Markov chain Monte Carlo, we demonstrate the broad applicability of this new programming pattern.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

Keywords Bayesian reasoning, machine learning, model-learner pattern, probabilistic programming

1. Introduction

Background: Bayesian Models and Inference We give the simple, general structure of a Bayesian model (see MacKay (2003) for instance); many examples follow later in the paper. Let y be the output of the model, such as the object to be predicted or observed, and let x be any input information on which to condition, such as the feature vector in classification or regression. Let w be the parameters of the model and let h be the hyperparameters. The key ingredients of a Bayesian model are the two conditional probability distributions:

- the *prior distribution* $p(w|h)$ over the parameters;
- the *sampling distribution* $p(y|x,w)$ over the output.

POPL'13, January 23–25, 2013, Rome, Italy.

The parameters w parameterize the sampling distribution over the output, while the hyperparameters h parameterize the prior distribution over the parameters.

Given training data $d = (x,y)$, we obtain by Bayes' rule expressions for computing the following distributions:

- the *posterior distribution* $p(w|d,h)$;
- the *posterior predictive distribution* $p(y'|x',d,h)$, assuming that (x',y') are independent from and identically distributed as (x,y).

This fundamental Bayesian model represents a wide variety of machine learning tasks. There are also a great many machine learning algorithms for *probabilistic inference*, that is, for computing exactly or approximately the posterior $p(w|d,h)$, and for using $p(y'|x',d,h)$ to make predictions.

Background: Probabilistic Programming for Inference Probabilistic programming language systems allow automatic generation of bespoke machine-learning algorithms. Such systems include AutoBayes (Schumann et al. 2008), Alchemy (Domingos et al. 2008), BUGS (Gilks et al. 1994), Church (Goodman et al. 2008), Csoft (Winn and Minka 2009), FACTORIE (McCallum et al. 2009), Figaro (Pfeffer 2010), Fun (Borgström et al. 2011), HANSEI (Kiselyov and Shan 2009), HBC (Daumé III 2008), IBAL (Pfeffer 2001), Probabilistic cc (Gupta et al. 1999), PFP (Erwig and Kollmansberger 2006), and Probabilistic Scheme (Radul 2007), amongst others.

The user writes a short probabilistic program, often embedded within a larger conventional program, and the system produces an algorithm for learning distributions given by the probabilistic program. Hence, probabilistic programming saves development costs compared to the alternative of writing the inference algorithm from scratch. Probabilistic programming is more flexible than the alternative of relying on a fixed algorithm for a particular task, as one can easily compose, refactor, and write variations of models. On the other hand, for some tasks a fixed algorithm may be more efficient, but much progress is being made on inference engines for probabilistic programs.

Still, the current practice of probabilistic programming is low-level, irregular, and unstructured. Probabilistic programs represent Bayesian models, but simply intermingle the code for defining parameters, predicting outputs, observing data, and so on. The absence of such structure is a missed opportunity. For example, in our own experience with Fun, we have over a dozen samples, but very little code re-use even though each sample performs similar tasks such as training, parameter learning, and prediction. We also duplicate code for the task of testing a model by sampling parameters, generating synthetic data from the predictive distribution, and then comparing with the outcome of learning the parameters from the

synthetic data. Moreover, there is repetition of common probabilistic patterns such as constructing mixture models. Other probabilistic programming systems share these problems.

Our Proposal: The Model-Learner Pattern The central idea of the paper is to add structure and code re-use to probabilistic programming by representing a Bayesian model by a generic type Model<TH,TW,TX,TY>. A value of this type is a record containing a hyperparameter together with probabilistic expressions for the prior and sampling distributions. The type parameters correspond to the constituents of a Bayesian model: hyperparameters h : TH, parameters w : TW, inputs x : TX, and outputs y : TY. The aim of the model-learner pattern is to make construction, use, and re-use of models easier (rather than to make inference more efficient). We are aware of no probabilistic system that encourages writing Bayesian models in a generic format.

Common patterns of constructing Bayesian models can be written as functions on these typed models. Given any model, we can derive a *sampler* object, which has general methods to draw samples from the prior and sampling distributions, for test purposes. Given any model and a suitable algorithm, we can derive a *learner* object, which has general methods to train on data, and to compute the posterior distribution and posterior predictive distributions.

Fun We evaluate the model-learner pattern by developing the idea in detail using Fun (Borgström et al. 2011), a probabilistic language embedded within F#, a dialect of ML for .NET. (The model-learner pattern can be developed in other probabilistic languages too, with or without types, but Fun has both a precise formal semantics and an efficient implementation.) As background, Section 2 recalls the syntax, type system, and informal semantics of Fun.

Model-Learner Pattern, and a Reference Learner Section 3 describes our notions of models, samplers, and learners using Fun, independently of any particular implementation of probabilistic inference. Theorem 1 asserts that the reference learner does indeed compute the posterior and posterior predictive distributions.

Learners based on Exact Inference and Factor Graphs We present in Section 4 a learner for discrete models that uses Algebraic Decision Diagrams (ADDs) as a compact representation of joint distributions. Theorem 2 shows correctness of the ADD backend. We show by example that Bayesian networks can be presented as models, and solved using ADDs, with performance comparable to other systems. Section 5 describes inference using probabilistic graphical models in Infer.NET (Minka et al. 2009). We describe our typed API for inferring distributions using Infer.NET.

Theory We enhance the previous semantics of Fun. The original version of Fun and its semantics (Borgström et al. 2011) did not include sums or observations on composite types, did not describe a semantics of Fun in the probability monad, and did not consider how to compute a density function for a probabilistic expression. In Section 6, we recall the measure-transformer semantics of Fun from previous work (Borgström et al. 2011). We extend Fun and the measure-transformer semantics with sum types, and give an inductive definition for observations on composite types. We identify a normal form with a single outermost observation, Theorem 3, enabling a semantics for many Fun expressions using the standard probability monad. These results provide a firm foundation for our semantics of the model-learner pattern.

Generic Mixtures Section 7 shows how generic compositions such as mixture models, evidence-based model averaging, and mixtures of experts can be seen as *model combinators*, that is, F# functions for producing models from models. Fun itself is first-order in the sense that functions are not values, but our combinators are higher-order functions in the host language F#. Theorem 4 shows that we can compute the evidence for a model (a measure of how well the model fits the observed data) using a simple if-expression, enabling a simple definition of these combinators and setting us apart from many other probabilistic languages.

Third Learner: via Markov chain Monte Carlo In Section 8, we develop the theory and implementation of a learner based on Markov chain Monte Carlo (MCMC) techniques. This learner is built on Filzbach (Purves and Lyutsarev 2012), a generic MCMC system. Given training data $d = (x, y)$, Bayesian MCMC computations explore different values for parameter w, and for each w require us to compute the density of the posterior function, that is, $p(w|d,h)$. Hence, to implement a learner, we need to compute the density of a probabilistic expression. We present a algorithm based on the direct symbolic evaluation of a probabilistic expression with only deterministic let bindings, verify its correctness as Theorem 5, and report our implementation. With our method, the user provides only a model, and our system automatically computes the density of the posterior; hence, we save the user effort compared to existing practice with Filzbach (and other MCMC systems) where the user provides code for both the sampling distribution and the density calculation.

Table of Model Types Table 1 at the end of the paper summarizes our collection of typed models. Our examples demonstrate that a wide range of tasks, including regression, classification, topic modelling, all fit the model-learner pattern. All our example models have been tested via at least one of our three learner backends. Our Infer.NET learner generates code of the same form as one would execute directly, so there is no performance penality. Our initial Filzbach learner used an interpretive log-posterior function, which was much less efficient than typical log-posterior functions written in C. Our current implementation now uses run-time code generation to compile the log-posterior function, yielding performance competitive with the C-code; further gains may be achieved by run-time specialization of this function to the data at hand. Our practical examples are evidence that a wide range of machine learning tasks are executable using the model-learning pattern, with no loss in performance in principle. but with the advantage of automatic synthesis of test data from the model, and in the case of MCMC-based systems such as Filzbach, the automatic construction of the density function.

Section 9 discusses related work and Section 10 concludes.

Contributions of the Paper The new conceptual insight is that code-based machine learning can be structured around *typed Bayesian models*, which are records containing a hyperparameter together with probabilistic expressions for prior and sampling distributions. Our specific technical contributions:

- Definition of a type of Bayesian models, with combinators for compositionally constructing models, and operations to derive samplers and learners from an arbitrary model.
- Many Bayesian examples expressed as such executable models.
- A formal semantics for models, learning, and prediction in terms of Fun, and its semantics using measure transformers and the probability monad.
- Learners based on Algebraic Decision Diagrams, message-passing on factor graphs, and MCMC.

Our generic format for models, and generic learner and sampler interfaces have several advantages over conventional probabilistic programming. An end-user may assemble new models from pre-existing models and combinators, without writing models from scratch. Our API is accessible from other languages such as C#. It is easy to add a new inference algorithm as a new learner. Finally, we enable generic programming for code re-use.

Most of the listings in the paper are directly imported from our F# code, and are a mixture of quoted probabilistic Fun code and deterministic functions in full F#.

A full version with additional details and proofs is available (Gordon et al. 2013).

2. Fun: Probabilistic Expressions (Review)

We recall the core calculus Fun (Borgström et al. 2011), enriched here with sum types to support the normal form of Section 6. Fun is a first-order functional language, without recursion. Our implementation efficiently supports a richer language with arrays and array comprehensions, and Vector and Matrix types, whose semantics can be seen as shorthands for constructs in this core.

We let c range over constant data of base and unit type, n over integers and r over real numbers. We write $\mathrm{ty}(c) = t$ to mean that constant c has type t.

Values and Types of Fun:

$U,V ::= x \mid c \mid (V,V) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$	value
$a,b ::= \mathbf{int} \mid \mathbf{real}$	base type
$t,u ::= \mathbf{unit} \mid (t_1 * t_2) \mid (t_1 + t_2)$	compound type

Let $\mathbf{bool} \triangleq \mathbf{unit} + \mathbf{unit}$. We let $\mathbf{V}_t$ be the set of closed values of type t (real numbers, integers, and so on). Semantically we consider **real** to be the reals, but our implementation uses double-precision floats. We assume a collection of deterministic functions on these types, including arithmetic and logical operators, and **fst** and **snd** projections on pairs. Each function f of arity n has a signature of the form $\mathbf{val}\ f\colon t_1 * \cdots * t_n \to t_{n+1}$. We assume standard families of primitive probability distributions of type $\mathsf{PDist}\langle t\rangle$, such as the following.

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

$\mathsf{Bernoulli} : (\mathsf{bias} : \mathbf{real}) \to \mathsf{PDist}\langle\mathbf{bool}\rangle$
$\mathsf{Beta} : (\mathsf{a} : \mathbf{real} * \mathsf{b} : \mathbf{real}) \to \mathsf{PDist}\langle\mathbf{real}\rangle$
$\mathsf{Gaussian} : (\mathsf{mean} : \mathbf{real} * \mathsf{prec} : \mathbf{real}) \to \mathsf{PDist}\langle\mathbf{real}\rangle$
$\mathsf{Gamma} : (\mathsf{shape} : \mathbf{real} * \mathsf{scale} : \mathbf{real}) \to \mathsf{PDist}\langle\mathbf{real}\rangle$

A Bernoulli distribution is a biased coin flip; the bias lies in the unit interval $[0,1]$ and is the probability of **true**. A Beta distribution, often used as a prior distribution on the bias of a Bernoulli distribution, is a distribution on the unit interval; when $\mathsf{a} = 1$ and $\mathsf{b} = 1$, it is the uniform distribution on the unit interval. A Gaussian distribution is parameterized by its mean and precision; the *standard deviation* σ follows from the identity $\sigma^2 = 1/\mathsf{prec}$. A Gamma distribution, often used as a prior distribution on the precision of a Gaussian distribution, is a distribution on the positive reals.

The expressions of Fun are in a syntax akin to administrative normal form, with **let**-expressions for sequential composition.

Expressions of Fun:

$M,N ::=$	expression
$\quad V$	value
$\quad f(V_1,\ldots,V_n)$	deterministic application
$\quad \mathbf{let}\ x = M\ \mathbf{in}\ N$	let (scope of x is N)
$\quad \mathbf{match}\ V\ \mathbf{with}\ \mathbf{inl}\ x : M \mid \mathbf{inr}\ y : N$	matching (scope of x is M, scope of y is N)
$\quad \mathbf{random}(\mathrm{Dist}(V))$	primitive distribution
$\quad \mathbf{observe}\ f(V_1,\ldots,V_n)$	observation
$\quad \mathbf{fail}$	failure

We rely on several standard syntactic conventions. We write $\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N$ for $\mathbf{match}\ V\ \mathbf{with}\ \mathbf{inl}\ \_ :\ M\ \mid \mathbf{inr}\ \_ :\ N$. We write $M;N$ for $\mathbf{let}\ x = M\ \mathbf{in}\ N$ when x is not free in N. Although formally Fun uses administrative normal form, we often allow arbitrary expressions M in places where values V are expected, assuming the insertion of suitable **let**-expressions. We allow arbitrary length tuples, formed by multiple pairings. We make use of records, and arrays and array comprehensions, where the size of each array is known statically; we consider operations on records and on statically-sized arrays to be reducible to operations on tuples.

Observation and failure expressions represent conditioning. We usually write observations in the form $\mathbf{observe}\ f(\overline{V})$, where $\overline{V}$ is short for $V_1,\ldots,V_n$. Such an observation expresses (unnormalized) conditioning on the event that the indicator function $f(\overline{V})$ yields a zero, defined as a closed value where every instance of c is 0. The primitive **fail** represents an impossible event. It has the same meaning as an observation such as $\mathbf{observe}(1+1)$ whose indicator function cannot yield zero. Note that **observe** is a no-op on **bool**, since any Boolean is a zero according to the above definition. We write $\mathbf{observe}\ (x = V)$ for $\mathbf{observe}\ (x - V)$ when $V = (c_1,\ldots,c_n)$ and $x - V$ is the component-wise difference.

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) the expression M has type t. We write $\diamond$ for the empty type environment. Apart from the following, the typing rules are standard.

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

$$\frac{\Gamma \vdash f(\overline{V}) : t}{\Gamma \vdash \mathbf{observe}\ f(\overline{V}) : \mathbf{unit}}\ (\textsc{Fun Observe}) \qquad \frac{\mathrm{Dist} : (x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle \quad \Gamma \vdash V : (t_1 * \cdots * t_n)}{\Gamma \vdash \mathbf{random}(\mathrm{Dist}(V)) : t}\ (\textsc{Fun Random})$$

Fun is designed as a subset of F#, so we represent Fun expressions using F#'s features for reflection (Syme 2006): quotation, evaluation, and antiquotation. We represent a closed term M of type t by the F# quotation <@ M @> of F# type Expr<t>. More generally, if $\diamond, x_1 : t_1,\ldots,x_n : t_n \vdash M : t$ we represent M by the F# quotation <@ **fun** $(x_1,\ldots,x_n) \to M$ @>. Suppose E is a quotation: E.Eval evaluates E to its value, and inside a quotation, the % symbol denotes antiquotation—the expression $(\%E)$ embeds the expression E within the quotation.

The semantics of Fun is detailed in Section 6; as usual, for precision concerning mixes of discrete and continuous probabilities, we turn to measure theory. Formally, the semantics of a closed Fun expression M is a finite measure $\mathscr{M}[\![M]\!]$ over its return type. A subtlety is that the semantics may or may not be a *probability distribution*, that is, one whose total measure is 1. Still, until Section 6, we think simply of all Fun expressions as defining probability distributions (implicitly normalising the measure), and use conventional mathematical notations for probability (as in the start of Section 1). We assume some familiarity with basic concepts such as joint distributions and marginalization.

Let an *inference engine* be an algorithm that given a quotation of a closed Fun expression M of type t returns a possibly approximate representation of the distribution $\mathscr{M}[\![M]\!]$. We can represent an inference engine as a function Expr<t> → DistT where DistT is the type of the representation.

3. Models, Samplers, and Learners

This section explains the central idea of the paper: that the prior and sampling distributions constituting a Bayesian model may be represented by a pair of typed Fun expressions. We introduce the distributions using conventional mathematical notations, alongside our formal notation using Fun expressions. We begin by explaining the idea for a specific example, and then generalize.

3.1 A Specific Bayesian Model: Linear Regression in Fun

As outlined in the introduction, a Bayesian model consists of a *prior distribution* $p(w|h)$ over parameter w, given hyperparameter h, together with a *sampling distribution* $p(y|x,w)$ over output y, given input information x and parameter w. Viewed as a function of the parameters w, with y fixed, the sampling distribution is also known as the *likelihood function*.

The prior distribution represents our uncertain beliefs about the parameters, given hyperparameter h. The sampling distribution represents our view of how output y is produced from input x given parameter w.

We consider the problem of linear regression, that is, of finding the best fitting line given a set of points. Our input data are $d = (x,y)$ where $x = [|x_1;\ldots;x_n|]$ and $y = [|y_1;\ldots;y_n|]$ are arrays of coordinates. Intuitively, we fit a noisy line $y_i = Ax_i + B + \varepsilon$ to the data, where the noise ε is drawn from a Gaussian distribution with *mean* 0 and *precision Prec*.

The expression prior h below expresses our initial uncertainty about the parameters A, B, and *Prec* of the noisy line, where hyperparameter h provides parameters for these distributions. (A standard alternative to a constant hyperparameter is to consider a prior over the hyperparameter, known as the *hyperprior*; we discuss hyperpriors in the full version.) The probabilistic expression gen(w,x) defines how to generate each y_i from x_i and parameters w. (In the Fun code below, we use record and array notations which are eventually treated as operations on tuples.)

The Prior and Sampling Distributions:

```
let prior h = { A = random (Gaussian(h.MeanA, h.PrecA))
                B = random (Gaussian(h.MeanB, h.PrecB))
                Prec = random (Gamma(h.Shape, h.Scale)) }
let gen (w,x) = [| for xi in x →
        (w.A * xi) + w.B + random(Gaussian(0.0, w.Prec)) |]
```

These expressions formalize the prior and sampling distributions. The prior distribution $p(w|h)$ is the density of $\mathscr{M}[\![\text{prior } h]\!]$ at w, while the sampling distribution $p(y|x,w)$ is the density of $\mathscr{M}[\![\text{gen}(w,x)]\!]$ at y. We assemble the two components of our model in the function predictive below, to obtain the *prior predictive distribution* $p(y|x,h)$, the density of $\mathscr{M}[\![\text{predictive}(h,x)]\!]$ at y.

The Prior Predictive Distribution:

```
let predictive (h,x) = let w = prior h in gen(w,x)
```

The semantics of the **let**-expression is obtained by forming the joint distribution of $\mathscr{M}[\![\text{prior } h]\!]$ and $\mathscr{M}[\![\text{gen}(w,x)]\!]$, and marginalizing with respect to $\mathscr{M}[\![\text{gen}(w,x)]\!]$. Formally, the semantics is obtained as the following integral.

$$p(y|x,h) = \int p(y|x,w)p(w|h)dw \tag{1}$$

We may sample from the prior predictive distribution by choosing h and input x as shown by the following F# output.

```
val h = {MeanA=0.0; PrecA=1.0; MeanB=5.0; PrecB=0.3;
         Shape=1.0; Scale=1.0}
val x = [|0.0; 1.0; 2.0; 3.0; 4.0; 5.0|]
```

We first sample w from the prior distribution $w =$ prior h, and then the output $y =$ gen(w,x).

```
val w = {A = 0.70; B = 4.33; Prec = 0.58;} // sampled prior(h)
val y = [|4.85; 5.19; 7.36; 4.49; 8.10; 8.06|] // sampled gen(w,x)
```

We express the posterior and posterior predictive distributions as Fun expressions, using **observe** to condition on the data (x,y).

The Posterior and Posterior Predictive Distributions:

```
let posterior (h,x,y) = let w = prior h in observe (y=gen(w,x)); w
let posterior_predictive(h,x,y,x') =
   let w = posterior (h,x,y) in gen(w,x')
```

Given observed data $d = (x,y)$, via Bayes' Rule we obtain a *posterior distribution*, that is, the prior in light of the data.

$$p(w|d,h) = \frac{p(y|x,w)p(w|h)}{p(d|h)} \tag{2}$$

The normalization constant $p(d|h) = \int p(y|x,w)p(w|h)dw$ is known as the *evidence* or *marginal likelihood*. We also obtain the *(posterior) predictive* distribution:

$$p(y'|x',d,h) = \int p(y'|x',w)p(w|d,h)dw \tag{3}$$

Using a particular inference engine for Fun we may obtain concrete representations of the normalized distributions. In the case of our running example, we try to infer the parameters used to generate our sample data y. By running our Infer.NET implementation of Fun to compute the distribution posterior(h,x,y) we obtain the following approximations.

```
{ A = Gaussian(0.5576, 0.05089); // actual A=0.70
  B = Gaussian(4.936, 0.404); // actual B=4.33
  Prec = Gamma(1.695, 0.46)[mean=0.78];} // actual Prec=0.58
```

Moreover, suppose we have new input $x' = [|6.0;7.0;8.0;9.0|]$. By running our Infer.NET implementation of Fun to compute the distribution posterior_predictive(h,x,y,x') we obtain the following.

```
[| Gaussian(8.282, 0.1858); Gaussian(8.839, 0.1654);
   Gaussian(9.397, 0.1469); Gaussian(9.954, 0.1303); |]
```

To summarize, we modelled a noisy line by distributions written as the Fun expressions prior h and gen(w,x). We ran these expressions to draw samples from the predictive distribution, so as to create a *synthetic dataset* $d = (x,y)$. We wrote Fun expressions for the posterior and posterior predictive distributions, and ran an inference engine to learn the posterior and to make predictions given fresh data. These tasks are the essence of Bayesian reasoning. The remainder of this section proposes generic types and interfaces for these tasks.

3.2 Typed Bayesian Models in General

In general, let a *typed Bayesian model* be a value of the record type Model<TH,TW,TX,TY>, where the type parameters correspond to the different data of a Bayesian model: hyperparameters h : TH, parameters w : TW, inputs x : TX, and outputs y : TY.

Typed Bayesian Model:

```
type Model<'TH,'TW,'TX,'TY> =
  { HyperParameter: 'TH
    Prior: Expr<'TH → 'TW>
    Gen: Expr<'TW * 'TX → 'TY> }
```

Given a model m, the Fun expression $h =$ m.HyperParameter is the hyperparameter, the Fun expression (%m.Prior) h is the prior distribution $p(w|h)$, and the Fun expression (%m.Gen)(w,x) is the sampling distribution $p(y|x,w)$. (We treat F# expressions such as (%m.Prior) h as being closed Fun expressions, although strictly speaking they are F# expressions that evaluate via antiquotation and function application to closed Fun expressions.)

The following module packages our linear regression code as a typed model M1. We use F# quotations <@...@> to treat the bodies of prior and gen above as Fun expressions.

Linear Regression Model:

```
module LinearRegression =
  type TH = {MeanA: real; PrecA: real; ... }
  type TW<'TA,'TB,'TN> = {A:'TA; B:'TB; Prec:'TN}
  type TX = real[]
  type TY = real[]
  let M1 = { HyperParameter = h
             Prior = <@ fun h → prior h @>
             Gen = <@ fun (w,x) → gen(w,x) @> }
         : Model<TH,TW<real,real,real>,TX,TY>
```

3.3 Sampling Parameters and Data

Given any model m, with $h =$ m.HyperParameter, we construct a new *sampler* S by sampling w from $p(w|h)$, and providing the methods:

- S.Parameters : TW is w sampled from the prior $p(w|h)$;
- S.Sample(x) : TY samples the sampling distribution $p(y|x,w)$.

The Sampler Interface:

```
type ISampler<'TW,'TX,'TY> = interface
    abstract Parameters: 'TW
    abstract Sample: x:'TX → 'TY
  end
```

Hence, a sampler draws from the prior predictive distribution. We omit our sampler code, which uses Eval to evaluate quotations.

3.4 Learning Parameters and Making Predictions

Given any model m, with $h =$ m.HyperParameter, and an inference engine, we may construct a new *learner* L with the following interface.

The Learner Interface:

```
type ILearner<'TDistW,'TX,'TY,'TDistY> = interface
    abstract Train: x:'TX * y:'TY →unit
    abstract Posterior: unit → 'TDistW
    abstract Predict: x:'TX → 'TDistY
  end
```

The type 'TDistW represents distributions over parameter 'TW, while the type 'TDistY represents distributions over output 'TY. Different learners may use different representations. Our ADD learner exactly represents distributions over Booleans using an ADD data structure, while our Infer.NET learner yields approximate parameters of the marginal distributions of each dimension of the distribution and our MCMC learner represents a distribution as an ensemble of samples.

We can think of a Fun quotation as an exact representation of a conditional distribution on its return type, independent of any inference engine. Using this idea, we present below our *reference learner*, which captures the intended exact semantics of our API, by assembling suitable quotations. The mutable variable d takes as values Fun expressions that represent the current parameter distribution, initially the posterior. Each call to Train updates d by conditioning with the training data. Calls to Posterior and Predict return suitable quotations for the posterior and posterior predictive distributions. (Compare with the posterior and posterior_predictive functions in Section 3.1.)

Reference Learner L for Model m:

```
let h = m.HyperParameter
let d = ref <@ (%m.Prior) h @>
{ new ILearner<Expr<'TW>,'TX,'TY,Expr<'TY>> with
  member l.Train(x:'TX,y:'TY) =
      d := <@ let w = (% !d) in
              observe(y = ((%m.Gen) (w,x)))
              w @>
  member l.Posterior():Expr<'TW> = (!d)
  member l.Predict(x:'TX):Expr<'TY> =
      <@ let w = (% !d) in (%m.Gen) (w,x) @> }
```

Theorem 1. *After n calls to* Train *with arguments* $d = \{(x_i, y_i)\}_{i=1}^n$,

- L.Posterior *represents the posterior distribution* $p(w|d,h)$*;*
- L.Predict(x') *represents the predictive distribution* $p(y'|x',d,h)$.

Our reference learner code is in fact the basis of our exact learner based on ADDs. Our other learners use numeric representations of intermediate distributions, rather than a quotation.

For example, a generic usage of our samplers and learners is to test whether an inference algorithm can recover known parameters from synthetic data. Consider a learner L constructed from a model m, hyperparameter h, and some inference engine. Given some input x, we may test the effectiveness of L by constructing a new sampler S for m and h, and running the following.

```
let w: TW = S.Parameters // fixed parameters
L.Train(x,S.Sample(x)) // train on synthetic data
let wD: TDistW = L.Posterior() // inferred distribution on w
... // test how probable w is according to wD
```

We abstract this pattern (as a function) in Section 3.6.

3.5 Generic Combinator for IID Models

If we assume that the data is a collection $d = \{(x_i, y_i)\}_{i=1}^n$ of identically and independently distributed (IID) observations, then the sampling distribution factorizes according to:

$$p(\{y_i\}_{i=1}^n | \{x_i\}_{i=1}^n, w) = \prod_{i=1}^{n} p(y_i|x_i, w) \tag{4}$$

Hence, we arrive at our first model combinator, IIDArray. Given a model that sends TX to TY, IIDArray builds a new model with the same prior, but which sends TX[] to TY[]. Learning with any model built from this combinator is an instance of *batch learning*, where multiple data items are processed simultaneously. (In the code below, we use an F# quotation of a **fun**-abstraction to represent a Fun expression with two free variables w and xs, as described in Section 2.)

Combinator to Lift Model to Act on Arrays:

```
module IIDArray =
  let M(m:Model<'TH,'TW,'TX,'TY>) =
    {HyperParameter = m.HyperParameter
     Prior = m.Prior
     Gen = <@ fun (w,xs) →[|for x in xs →(%m.Gen)(w,x)|] @> }
    : Model<'TH,'TW,'TX[],'TY[]>
```

For example, M2 below is linear regression on a single (x, y) point, and M3, obtained by our combinator, is equivalent to our original model M1. Models in the style of M2 are useful because we may assemble them with other combinators before applying IIDArray, as shown in Section 7.

Linear Regression, Again and Again:

```
let M2 =
  { HyperParameter = h
    Prior = <@ fun h → prior h @> // as before
    Gen = <@ fun(w,x) →
      (w.A ∗ x) + w.B + random(Gaussian(0.0,w.Prec)) @>
  }: Model<TH,TW<real,real,real>,real,real>
let M3 = IIDArray.M(M2)
```

3.6 Generic Loopback Testing

A common method to test the effectiveness of a learner on a particular model is to generate IID sample data for different x_i for a fixed w, and then evaluate the posterior distribution for w obtained by training on that data. We here give a generic procedure for such a loopback test.

Functions for loopback testing:

```
let test (toLearner: Model<'TH,'TW,'TX,'TY> →
                     ILearner<'DistW,'TX,'TY,'DistY>)
         (m:Model<'TH,'TW,'TX,'TY>)
         (x:'TX) : 'TW ∗ 'DistW =
  let L = toLearner(m)
  let S = Sampler.FromModel(m)
  let y = S.Sample(x)
  do L.Train(x,y)
  (S.Parameters,L.Posterior())
let testMany l m xs = test l (IIDArray.M(m)) xs
```

Since the details of evaluating the inferred posterior depend on its learner-specific representation, test simply returns the parameters and posterior distribution to the caller. Notice how testMany is constructed by a simple application of the model combinator IIDArray.M.

Here is an application of testMany to our running example, linear regression, using the Infer.NET learner we describe below:

Loopback testing of linear regression:

```
let toL m = InferNetLearner.LearnerFromModel(m,Marginalize)
let (w,dW) = testMany toL M2 [| −100.0 .. 1.0 .. 100.0 |]
```

On one test run, we obtained:

```
w= { A = 0.3477837603;
     B = −27.13593566;
     Prec = 2.103660906;}
dW={ A = Gaussian(0.3479, 1.467e06);
     B = Gaussian(−27.09, 435.0);
     Prec = Gamma(99.74, 0.02203)[mean=2.197];}
```

Here the learner has inferred close approximations to A, B and Prec, with very high precision especially for A.

4. A Learner using Exact Inference

The exact learner uses symbolic evaluation of Boolean functions representing discrete distributions in order to give an answer that is either an *exact* marginal of each variable or the *exact* joint distribution. Let Bernoulli Fun be the finite fragment of Fun where the only sum type is $\mathbf{bool} = \mathbf{unit} + \mathbf{unit}$, the only matches are conditionals, there are no instances of **observe**, and every **random** expression takes the form **random** $(\mathsf{Bernoulli}(c))$ for some real $c \in (0,1)$. The semantics of a closed Bernoulli Fun program is a sub-probability distribution over its discrete return type, that is, a measure that may sum to less than 1.

4.1 Inference by Symbolic Evaluation of Programs

The procedure POST symbolically evaluates a Bernoulli Fun expression M, and computes the posterior sub-distribution over the result of M given a valuation of its free variables. The algorithm relies on the following notations. We write **let** $x : t = N$ **in** M for the corresponding untyped **let**-expression when $\Gamma \vdash N : t$. We write $\mathtt{ite}(v = U, V_T, V_F)$ for the conditional statement that returns V_T when $v = U$ and V_F otherwise. A valuation σ is a finite map from variables to closed values; A Γ-valuation σ is a finite map $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$ where each $V_i \in \mathbf{V}_{t_i}$. We let $\diamond$ denote the empty valuation, and write $S(\Gamma)$ for the set of Γ-valuations. If $\Gamma \vdash V : t$ and σ is a Γ-valuation, $V\sigma$ is the closed value resulting from substituting $\sigma(x)$ for x in V for all $x \in \mathrm{dom}(\Gamma)$. Given an open expression M, where $\Gamma \vdash M : t$, POST is a function from valuations $\sigma \in S(\Gamma)$ to discrete sub-probability distributions over $\mathbf{V}_t$. By uncurrying, the semantics of M can also be expressed as a function from $S(\Gamma) \times \mathbf{V}_t$ to sub-probabilities $[0,1]$. The algorithm POST (M) computes this function by structural recursion on M. The POST function is a forward analysis that computes the posterior distribution, and is named by analogy with the standard post-condition computation in program analysis.

Post Calculation POST $(M) : S(\Gamma) \times \mathbf{V}_t \to$ **real when** $\Gamma \vdash M : t$

$$\textsc{Post}(V) = \lambda\sigma, v.\mathtt{ite}(v = V\sigma, 1, 0)$$

$$\textsc{Post}(f(V_1, \ldots, V_n)) = \lambda\sigma, v.\mathtt{ite}(v = f(V_1\sigma, \ldots, V_n\sigma), 1, 0)$$

$$\textsc{Post}(\mathsf{Bernoulli}(r)) = \lambda\sigma, v.\mathtt{ite}(v = \mathbf{true}, r, 1-r)$$

$$\textsc{Post}(\mathbf{if}\ V\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2) = \lambda\sigma, v.\textsc{Post}(V)[\sigma, \mathbf{true}] \cdot \textsc{Post}(N_1)[\sigma, v] + \textsc{Post}(V)[\sigma, \mathbf{false}] \cdot \textsc{Post}(N_2)[\sigma, v]$$

$$\textsc{Post}(\mathbf{let}\ x : t = N_1\ \mathbf{in}\ N_2) = \lambda\sigma, v.\textstyle\sum_{u \in \mathbf{V}_t} \textsc{Post}(N_1)[\sigma, u] \cdot \textsc{Post}(N_2)[(\sigma, x \mapsto u), v]$$

$$\textsc{Post}(\mathbf{fail}) = \lambda(\sigma, ()).0$$

The POST computation uses four different types of operations on functions: (1) pointwise product, (2) pointwise sum, (3) if-then-else, (4) summation over a variable, or existential quantification. The operations (1) and (2) are used, for example, in the case for **if**, operation (3) is used in the cases for values, deterministic functions, and Bernoulli, and operation (4) is used in the **let** case. Each of these operations is directly supported by ADD packages such as CUDD (Somenzi 2012) and takes time proportional to the product of the sizes of the arguments in the worst case.

Theorem 2. *Let M be a Bernoulli Fun program with $\diamond \vdash M : t$. For all values $V \in \mathbf{V}_t$, $\textsc{Post}(M)[\diamond, V] = \mathscr{M}[\![M]\!]\{V\}$.*

The ADD learner builds on the reference learner, calling the POST algorithm on the Fun expression returned from its Posterior and Predict methods. An ADD learner has the following type, where ADD<'T> is the type of a decision diagram that maps values of type 'T to probabilities.

```
: ILearner<ADD<'TW>, 'TX, 'TY, ADD<'TY>>
```

In our experiments, the performance of the ADD learner is competitive with algorithms implemented in SamIam (Darwiche 2009). Claret et al. (2012) report performance in detail for symbolic evaluation of a related imperative probabilistic language.

4.2 Example: Bayes Net for Sprinkler

The sprinkler model is a classical example of a Bayes net (Pearl and Shafer 1995). In our variant, the model describes the conditional probabilities of rain having fallen and the sprinkler having been on, given that the grass is observed to be wet. In detail, given prior distributions for rain and sprinkling, and the view that rain has a

90% chance of wetting the grass, sprinkling an 80% chance, and some other cause a 10% chance, what is the posterior distribution for rain and sprinkling, given that the grass is wet. We perform inference using the ADD learner described above, yielding an exact posterior.

Sprinkler model

```
module Sprinkler =
  type TH = {RainH: real; SprinklerH: real}
  type TW<'b> = {Rain: 'b; Sprinkler: 'b}
  type TX = IsGrassWet // a unit type
  type TY = bool
  let M:Model<TH,TW<bool>,TX,TY> =
   {HyperParameter = {RainH=0.3; SprinklerH=0.5}
    Prior = <@ fun h →
     {Rain = random(Bernoulli(h.RainH))
      Sprinkler = random(Bernoulli(h.SprinklerH))} @>
    Gen = <@ fun (w,x) →
      (random (Bernoulli(0.9)) && w.Rain) ||
      (random (Bernoulli(0.8)) && w.Sprinkler) ||
      random (Bernoulli(0.1)) @>}
```

Given an ADD learner L for this model, here is the outcome of L.Train(IsGrassWet, **true**); L.Posterior.

```
[({Rain = false; Sprinkler = false;}, 0.05777484318);
 ({Rain = true; Sprinkler = false;}, 0.2253218884);
 ({Rain = false; Sprinkler = true;}, 0.4737537141);
 ({Rain = true; Sprinkler = true;}, 0.2431495543)];
```

5. A Learner based on Factor Graph Inference

Infer.NET is a probabilistic programming system which generates efficient, scalable inference algorithms based on message-passing on factor graphs. We compile a Fun expression to the input format of Infer.NET, Csoft, and perform inference (Borgström et al. 2011).

Since the original description of Fun, we have made a series of enhancements which make the present paper possible. These include the API described next, support for arrays, and a compiler based on multiple transformations of the Fun expression.

5.1 Learner for Message-Passing Algorithms in Infer.NET

We describe our interface for creating learners based on Infer.NET, but omit the details of the implementation, which rely on the existing translation from Fun to Infer.NET.

Infer.NET computes marginal posterior distributions from joint distributions involving observed and unobserved variables. For example, given a probabilistic program defining a joint distribution on pairs of type **bool** × **real**, Infer.NET computes the marginal of the first projection as a value d_1 say of the distribution type Bernoulli, and computes the approximate marginal of the second projection as a value d_2 say of type Gaussian. (Bernoulli and Gaussian are Infer.NET types representing the distribution as a record of its parameters.) Infer.NET uses the distribution families of the priors on these projections as (approximate) distribution types for the marginals. Let a *marginal type G* be a nesting of tuples, records and arrays, over distribution types such as Bernoulli and Gaussian. Let joint(G) be G with each occurrence of an Infer.NET type replaced with its range; for instance, joint(Bernoulli) = **bool** and joint(Gaussian) = **real**.

We perform inference using Infer.NET using the following function, where CompoundDistribution is a dynamically typed value representing the composite of the marginal distributions.

Core Inference for Infer.NET Fun:

```
val inferDynamic: Expr<'TA → 'TB> →
                  'TA → CompoundDistribution
```

Next, we describe use of automatic coercions to achieve better static typing than with inferDynamic. A *compound distribution for G* is a value of CompoundDistribution representing a value of type G. A *G-coercion* is a function CompoundDistribution → G that coerces a compound distribution for G to the corresponding value.

To create a learner for m : Model<TH,TW,TX,TY> we ask the user to supply F# marginal types GW and GY such that joint(GW) = TW and joint(GY) = TY, plus a MarginalizeModel<GW,GY>, which is a pair of a GW-coercion and a GY-coercion. (We have helper functions to construct these coercions, but we omit the details.) The resulting learner does approximate inference on the model m. After each call to Train, we run inference and record the result at marginal type GW. Calling Posterior returns the current distribution. Calling Predict runs inference, and we return the result at marginal type GY.

Constructing an Infer.NET Learner:

```
type MarginalizeModel<'GW,'GY> =
  { MarginalizePrior: CompoundDistribution → 'GW
    MarginalizeGen: CompoundDistribution → 'GY }

val Learner: (Model<'TH,'TW,'TX,'TY> * 'TH
            * MarginalizeModel<'GW,'GY>
           → ILearner<'GW,'TX,'TY,'GY>
```

5.2 Example: Gaussian

Here is the model for the ubiquitous Gaussian. The parameters are its Mean and Precision. Their priors are a Gaussian and a Gamma. In turn, the hyperparameter consists of the parameters for the priors. The sampling distribution simply draws from the Gaussian, ignoring the input of type TX = **unit**. (A model that ignores its input is said to be *unsupervised*.)

Gaussian Model:

```
module GaussianModel =
 type GammaW<'TA,'TB> = {Shape: 'TA; Scale: 'TB}
 type TW<'TA,'TB> = {Mean:'TA; Precision:'TB}
 type TH = { Gaussian: TW<real,real>
             Gamma: GammaW<real,real> }
 let M: Model<TH,TW<real,real>,unit,real> =
  {HyperParameter = {Gaussian={Mean=0.0;Precision=1.0}
                     Gamma={Shape=1.1;Scale=2.0}}
   Prior= <@ fun h → {Mean= let m = h.Gaussian.Mean
                           let p = h.Gaussian.Precision
                           random(Gaussian(m,p))
                      Precision=let sh = h.Gamma.Shape
                                let sc = h.Gamma.Scale
                                random(Gamma(sh,sc))}@>
   Gen= <@ fun (w,x) →let m,p = w.Mean,w.Precision
                      random(Gaussian(m,p)) @> }
```

We obtain a learner of the following type. The subject of the marginal type for y, Gaussian, is **real**, while the subject of the marginal type for w, TW<Gaussian,Gamma>, is TW<**real**,**real**>. (We make TW generic so it can express both these types).

```
: ILearner<TW<Gaussian,Gamma>,unit,real,Gaussian>
```

After inference, we can inspect the parameters of the inferred Gaussian and Gamma distributions to learn the inferred mean and variance of the data, and the remaining uncertainty thereof.

Probit is a standard binary classifier built from a Gaussian. Its model shares the same prior, and its output is a probabilistic Boolean dependent on its input. We use this model in Section 7.

Probit Model:

```
let Probit : Model<TH,TW<real,real>,real,bool> =
  {HyperParameter = M.HyperParameter
   Prior = M.Prior
   Gen = <@ fun (w,x) →
            x < random(Gaussian(w.Mean,w.Precision)) @> }
```

We have a range of other models including multivariate linear regression, the Bayes Point Machine classifier (Minka 2001), the LDA (Latent Dirichlet Allocation) topic modelling (Blei et al. 2003), and TrueSkill (Herbrich et al. 2006), but we omit the details.

6. Semantics of Models: Measures and Monads

In this section, we review the measure-transformer semantics of Fun (Borgström et al. 2011), and extend it with sum types and observations on composite types. Based on our experience with Fun models, their structure often includes a single outermost **observe** statement. Here, we show that given a certain *compatibility* constraint on its semantics, any Fun expression can be transformed to one with a single outermost **observe**, preserving the semantics. As part of the proof, we show that the measure-transformer semantics of **observe**-free Fun models is closely related to their semantics when considered as programs of the stochastic lambda-calculus of Ramsey and Pfeffer (2002). This clarifies the relationship between our measure-transformer semantics and the probability monad, in particular for programs with none or only a single outermost observation.

6.1 Measure Transformer Semantics for Fun Expressions

In this section, we recall the measure-transformers used in Borgström et al. (2011), and augment these with operations on sum types and failure. We also give an inductive definition of the semantics of observations on composite types, and recall the compositional denotational semantics of a Fun expression as a measure transformer, augmented with sum types. For more explanations and intuitions, see (Borgström et al. 2011). We conclude by introducing the notion of *compatible* measure, and show that it is sufficient (but not necessary) for all observations to be well-defined.

We define the measurable sets of type t, written $\mathscr{M}_t$, as the Lebesgue-measurable subsets of $\mathbf{V}_t$, which in particular contains all closed sets. We write $f : t \to u$ to mean that f is a measurable function from type t to type u, that is, that $f^{-1}(A) \in \mathscr{M}_t$ for all $A \in \mathscr{M}_u$. Let $\mathtt{M}\,t$ be the set of finite measures on t, that is, additive functions from $\mathscr{M}_t$ to the non-negative real numbers. Let the sub-probability distributions $\mathtt{S}\,t$ be the finite measures whose range is contained in $[0,1]$. Let $t \rightsquigarrow u$ be the set of measure transformers from t to u, defined as the partial functions $\mathtt{M}\,t \rightharpoonup \mathtt{M}\,u$. If $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ we let $\mathrm{range}(\Gamma) \triangleq t_1 * \cdots * t_n$. We make use of the following constructions on measures.

- The Dirac δ measure is $\delta_V(A) \triangleq 1$ if $V \in A$, 0 otherwise.
- Given a function $f : t \to u$ and a measure $\mu \in \mathtt{M}\,t$, there is a measure $\mu f^{-1} \in \mathtt{M}\,u$ given by $(\mu f^{-1})(B) \triangleq \mu(f^{-1}(B))$. We can add two measures on the same set as $(\mu_1 + \mu_2)(A) \triangleq \mu_1(A) + \mu_2(A)$. The disjoint sum $(\mu_1 \uplus \mu_2)$ of two measures is defined as $(\mu_1 \uplus \mu_2)(A \uplus B) = \mu_1(A) + \mu_2(B)$.
- Given a measure μ on t, a measurable set $A \in \mathscr{M}_t$ and a function $f : t \to \mathbf{real}$, we write $\int_A f\,d\mu$ or equivalently $\int_A f(x)\,d\mu(x)$ for standard (Lebesgue) integration. This integration is always well-defined if μ is finite and f is bounded.
- Given a measure μ on t, a function $D\mu : t \to \mathbf{real}$ is a *density* for μ iff $\mu(A) = \int_A (D\mu)\,d\lambda$ for all A, where λ is the completion of the standard measure on t (which is built by taking products and disjoint sums of the counting measure on **int** and the interval (Borel) measure on **real**).

The semantics of a Fun program is given in terms of the following measure transformers, which encapsulate standard theorems in finite measure theory.

Measure Transformer Combinators:

$\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$
$\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$
$\mathtt{extend} \in (t \to \mathtt{S}\,u) \to (t \rightsquigarrow (t * u))$
$\mathtt{observe} \in (t \to \mathbf{real}) \to (t \rightsquigarrow t)$
$\parallel\!\!\parallel \in (t_1 \rightsquigarrow u) \to (t_2 \rightsquigarrow u) \to ((t_1 + t_2) \rightsquigarrow u)$
$\mathtt{fail} \in (t \rightsquigarrow t)$

To lift a pure measurable function to a measure transformer, we use the combinator `pure`. Given $f : t \to u$, we let $\mathtt{pure}\ f\ \mu\ A \triangleq \mu f^{-1}(A)$, where $\mu \in \mathtt{M}\,t$ and A is a measurable set from u.

To sequentially compose two measure transformers we use standard function composition, defining $T \ggg U \triangleq U \circ T$.

The combinator `extend` extends the domain of a measure using a function yielding sub-probability distributions. We let $\mathtt{extend}\ m\ \mu\ AB \triangleq \int_{\mathbf{V}_t} m(x)(\{y \mid (x,y) \in AB\})\,d\mu(x)$.

The combinator `observe` computes the conditional density of a measure μ over t on the event that an indicator function p of type $t \to \mathbf{real}$ is zero. Note that this conditioning is *unnormalized*. We consider the family of events $p(x) = r$ where r ranges over $\mathbb{R}$. We let $\mathbf{B}^r_\varepsilon$ be the closed ball of radius ε around r (that is, $[x-\varepsilon, r+\varepsilon]$), and define $\dot{\mu}[A||p = r] \in \mathbb{R}$ (the μ-density at $p = r$ of A) by:

Conditional Density: $\dot{\mu}[A||p = r]$

$\dot{\mu}[A||p = r] \triangleq \lim_{\varepsilon \to 0} \mu(A \cap p^{-1}(\mathbf{B}^r_\varepsilon)) / \int_{\mathbf{B}^r_\varepsilon} 1\,d\lambda$ if the limit exists

We define $\mathtt{observe}\ p\ \mu\ A \triangleq \dot{\mu}[A||p = 0]$. As an example, if $t = u * \mathbf{real}$, $p = \lambda(x,y).(y - c)$ and μ has continuous density $D\mu$ then

$$\mathtt{observe}\ p\ \mu\ A = \int_{\{x \mid (x,c) \in A\}} D\mu(x,c)\,d\lambda(x) \tag{5}$$

and $\int_{\mathbb{R}} \dot{\mu}[A||p = x]\,d\lambda(x) = \mu(A)$. Notice that $\mathtt{observe}\ p\ \mu\ A$ is greater than $\mu(A)$ if the density at $p = 0$ is greater than 1, so we cannot consider only transforming (sub-)probability distributions.

Support for sum types (new) To add support for sum types, we introduce a new measure transformer. We let $(T_1 \parallel\!\!\parallel T_2)\ \mu\ A \triangleq T_1(X \mapsto \mu(X \uplus \varnothing))(A) + T_2(X \mapsto \mu(\varnothing \uplus X))(A)$. We also let $\mathtt{fail}\ \mu\ A \triangleq 0$.

Observations on composite types (new) The Fun language of Borgström et al. (2011) permits observations only on atomic types. To achieve a normal form (Theorem 3) we here extend observations to composite types. We write $\mathtt{obs}\ f@u$ if $f : t \to u$; then the measure transformer $\mathtt{obs}\ f$ has type $t \rightsquigarrow t$. Below we let $1_b \triangleq 1$, $1_{\mathbf{unit}} \triangleq ()$, $1_{t*u} \triangleq (1_t, 1_u)$ and $1_{t+u} \triangleq 1_t$. We write $\mathtt{split}\ p = \mathtt{pure}\ \lambda s.\mathtt{if}\ p(s)\ \mathtt{then}\ \mathbf{inl}\ s\ \mathtt{else}\ \mathbf{inr}\ s$.

Observations on Composite Types: $\mathtt{obs}\ f@u$

$\mathtt{obs}\ f@\mathbf{real} \triangleq \mathtt{observe}\ f$
$\mathtt{obs}\ f@\mathbf{int} \triangleq \mathtt{split}\ (\lambda s.f(s) = 0) \ggg (\mathtt{pure\ id} \parallel\!\!\parallel \mathtt{fail})$
$\mathtt{obs}\ f@\mathbf{unit} \triangleq \mathtt{pure\ id}$
$\mathtt{obs}\ f@(u_1 * u_2) \triangleq \mathtt{obs}\ (\mathtt{snd} \circ f)@u_2 \ggg \mathtt{obs}\ (\mathtt{fst} \circ f)@u_1$
$\mathtt{obs}\ f@(u_1 + u_2) \triangleq \mathtt{split}\ f \ggg$
$\qquad (\mathtt{obs}\ (\mathtt{either\ id}\ \lambda\_.1_{u_1}) \circ f@u_1$
$\qquad \parallel\!\!\parallel\ \mathtt{obs}\ (\mathtt{either}\ \lambda\_.1_{u_2}\ \mathtt{id}) \circ f@u_2)$

Observation at **real** type is primitive; it is trivial at **unit** type. At **int** we `fail` unless f returns 0. For products, we observe each component in turn. For sum types, we split the set of states depending on the branch of the sum picked by f, and run the observation in each branch. Note that observations on discrete types yield a series of `splits` where some branches `fail`, and that $\texttt{obs}\ \lambda\_.1@\textbf{int} = \texttt{fail}$.

Measure transformer semantics We can then give a compositional semantics of a Fun program as a measure transformer, taking a measure over assignments to its free variables and returning a joint measure over variable assignments and return value. To bind the values of a valuation to the corresponding variables we use $\mathrm{pat}(\diamond) = ()$ and $\mathrm{pat}(xs,x) := (\mathrm{pat}(xs),x)$ as patterns and in return values. Below we define the measure transformer semantics of a program as $\mathscr{A}[\![M]\!]_{xs}$ where xs are the free variables in the program. We use an auxiliary definition $\mathscr{A}[\![M]\!]^{y}_{xs}$ for the semantics of the program M in the scope of y. For closed terms M we obtain $\mathscr{M}[\![M]\!]$ by transforming the trivial probability measure.

Each signature **val** $f\colon t_1 * \cdots * t_n \to t_{n+1}$ means that f is a total function with $f : \mathbf{V}_{t_1 * \cdots * t_n} \to \mathbf{V}_{t_{n+1}}$. If $\mathrm{Dist} : t \to \mathsf{PDist}\langle u\rangle$ and $V \in \mathbf{V}_t$, there is a corresponding measure $\mu_{\mathrm{Dist}(V)}$ on $\mathbf{V}_u$. This measure is a probability measure for legal values of V, and otherwise, such as in $\mathsf{Bernoulli}(3.0)$, it is the zero measure.

To apply $|||$ below we need to lift a sum to the top level of a type. We do this with $\texttt{lift1} : t*(u_1+u_2) \to (t*u_1)+(t*u_2)$ defined as $\lambda x,y.\texttt{match}\ y\ \texttt{with}\ \texttt{inl}\ z : \texttt{inl}(x,z)\ \mid\ \texttt{inr}\ z : \texttt{inr}(x,z)$.

Measure Transformer Semantics of Fun: $\mathscr{M}[\![M]\!]$**,** $\mathscr{A}[\![M]\!]_{xs}\ \mu$

$$\mathscr{M}[\![M]\!]\ A \triangleq \mathscr{A}[\![M]\!]_{\diamond}\ \delta_{()}\ \{()\} \times A$$

$$\mathscr{A}[\![M]\!]^{y}_{xs} \triangleq \mathscr{A}[\![M]\!]_{xs,y} \ggg \texttt{pure}\ \lambda(\mathrm{pat}(xs,y),z).(\mathrm{pat}(xs),z)$$

$$\mathscr{A}[\![V]\!]_{xs} \triangleq \texttt{pure}\ \lambda\,\mathrm{pat}(xs).(\mathrm{pat}(xs),V)$$

$$\mathscr{A}[\![f(V_1,\ldots,V_n)]\!]_{xs} \triangleq \texttt{pure}\ \lambda\,\mathrm{pat}(xs).(\mathrm{pat}(xs),f(\overline{V}))$$

$$\mathscr{A}[\![\textbf{match}\ V\ \textbf{with inl}\ x :\ M\ \mid \textbf{inr}\ y :\ N]\!]_{xs} \triangleq \mathscr{A}[\![V]\!]_{xs} \ggg \texttt{pure lift1} \ggg (\mathscr{A}[\![M]\!]^{x}_{xs} \mid\!\mid\!\mid \mathscr{A}[\![N]\!]^{y}_{xs})$$

$$\mathscr{A}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!]_{xs} \triangleq \mathscr{A}[\![M]\!]_{xs} \ggg \mathscr{A}[\![N]\!]^{x}_{xs}$$

$$\mathscr{A}[\![\textbf{random}(\mathrm{Dist}(V))]\!]_{xs} \triangleq \texttt{extend}\ \lambda\,\mathrm{pat}(xs).\mu_{\mathrm{Dist}(V)}$$

$$\mathscr{A}[\![\textbf{observe}\ f(\overline{V})]\!]_{xs} \triangleq \texttt{obs}\ \lambda\,\mathrm{pat}(xs).f(\overline{V}) \ggg \mathscr{A}[\![()]\!]_{xs}$$

$$\mathscr{A}[\![\textbf{fail}]\!]_{xs} \triangleq \texttt{fail} \ggg \mathscr{A}[\![()]\!]_{xs}$$

Note that $\mathscr{A}[\![\textbf{observe}\ 1+1]\!]_{xs} = \mathscr{A}[\![\textbf{fail}]\!]_{xs}$. Indeed, **fail** suffices to implement all observations on discrete types (by inlining `obs`, implementing `split` using **if**).

Proposition 1 (Static Adequacy)**.**
If $\Gamma \vdash M : t$ *then* $\mathscr{A}[\![M]\!]_{\mathrm{dom}(\Gamma)} \in \mathrm{range}(\Gamma) \rightsquigarrow (\mathrm{range}(\Gamma) \times \mathrm{t})$.

As seen above, $(\texttt{observe}\ p)\ \mu$ might be undefined, if its defining limit does not exist. We here give a sufficient condition for the limit to exist. We inductively define compatibility of a measure with respect to all the observations in a composition of measure transformer combinators as follows. We write $\mu \models T$ for "μ is compatible with respect to T".

Compatibility: $\mu \models T$ **(**$T \in t \rightsquigarrow u$ **and** $\mu \in \mathtt{M}\ t$**)**

$\mu \models \texttt{pure}\ f$ $\qquad$ $\mu \models \texttt{extend}\ m$ $\qquad$ $\mu \models \texttt{fail}$

$\mu_1 \uplus \mu_2 \models T_1 \mid\!\mid\!\mid T_2$ $\quad$ iff $\mu_1 \models T_1$ and $\mu_2 \models T_2$

$\mu \models T \ggg U$ $\quad$ iff $\mu \models T$ and $(T\ \mu) \models U$

$\mu \models \texttt{observe}\ p$ $\quad$ iff p is an affine function, and there is ε such that $D\,\mu$ is continuous on $p^{-1}(\mathbf{B}^0_\varepsilon)$ and p is not constant on any open subset of $p^{-1}(\mathbf{B}^0_\varepsilon)$.

Proposition 2. *If* $\mu \models T$*, then* $T\,\mu$ *is defined.*

Compatibility is not necessary for $T\,\mu$ to be defined:

Hybrid Measure:

```
let hybrid1 = if random(Bernoulli(0.5)) then 1.0
              else random(Gaussian(0.0,1.0))
let hybrid2 = (random(Gaussian(0.0,r)),hybrid1)
```

The measure $\mu = \mathscr{M}[\![\mathsf{hybrid2}]\!]$ is the average of a two-dimensional Gaussian distribution and a line mass at $y = 1.0$. Here $\mu \models \texttt{observe}\ \lambda(x,y).y$, but we do not have $\mu \models \texttt{observe}\ \lambda(x,y).x$ because the line $x = 0.0$ crosses the line mass at $y = 1.0$, where the density fails to exist. However, by the definition of `observe` we get $(\texttt{observe}\ \lambda(x,y).x)\ \mathscr{M}[\![\mathsf{hybrid2}]\!] = \sqrt{r}\cdot\phi(0.0)\cdot\mathscr{M}[\![(0.0,\mathsf{hybrid1})]\!]$ where $\phi(0.0) \approx 0.3989$ is the probability density of the standard normal distribution at 0.0. Note that as the precision r above grows, the weight of the resulting distribution grows, with no upper bound.

6.2 Monadic Semantics for Fun with fail

If M does not contain any occurrence of **observe** or **fail** then M is a term in the language of (Ramsey and Pfeffer 2002) which has a semantics using the probability monad (Giry 1982). To treat **fail** we work in the sub-probability monad (Panangaden 1999), where the set of admissible distributions μ also admits $|\mu| < 1$ (cf. the semantics of Barthe et al. (2012)).

The valuation σ maps the free variables of M to closed values.

Monadic Semantics of Fun with fail: $\mathscr{P}[\![M]\!]\ \sigma$

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

$$\mathscr{P}[\![V]\!]\ \sigma \triangleq \texttt{return}\ (V\sigma)$$

$$\mathscr{P}[\![f(V_1,\ldots,V_n)]\!]\ \sigma \triangleq \texttt{return}\ f(V_1\sigma,\ldots,V_n\sigma)$$

$$\mathscr{P}[\![\textbf{match}\ V\ \textbf{with inl}\ x :\ M\ \mid \textbf{inr}\ y :\ N]\!]\ \sigma \triangleq \mathscr{P}[\![V]\!]\ \sigma \gg\!= \texttt{either}\ (\lambda v.\mathscr{P}[\![M]\!]\ (\sigma, x \mapsto v))\ (\lambda v.\mathscr{P}[\![N]\!]\ (\sigma, y \mapsto v))$$

$$\mathscr{P}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg\!= \lambda v.\mathscr{P}[\![N]\!]\ (\sigma, x \mapsto v)$$

$$\mathscr{P}[\![\textbf{random}(\mathrm{Dist}(V))]\!]\ \sigma \triangleq \mu_{\mathrm{Dist}(V\sigma)}$$

$$\mathscr{P}[\![\textbf{fail}]\!]\ \sigma \triangleq \texttt{zero}$$

Proposition 3. *If* $x_1 : t_1, \ldots, x_n : t_n \vdash M : t$ *and* $xs = x_1, \ldots, x_n$ *and all observations in* M *are* **fail***, then*
$\mathscr{A}[\![M]\!]_{xs} = \texttt{extend}\ \lambda(v_1,\ldots,v_n).\mathscr{P}[\![M]\!]\ (x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$.

In particular, if $\diamond \vdash M : t$ we have $\mathscr{M}[\![M]\!] = \mathscr{P}[\![M]\!]\ \diamond$.

6.3 A Normal Form for Fun Expressions

Observation Translation: $\mathrm{O}[\![M]\!]$ $\qquad$ **(**o **does not appear in** N**)**

$$\mathrm{O}[\![\textbf{observe}\ f(\overline{V})]\!] \triangleq (), f(\overline{V})$$

$$\mathrm{O}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] \triangleq \textbf{let}\ x, o = \mathrm{O}[\![M]\!]\ \textbf{in let}\ y, o' = \mathrm{O}[\![N]\!]\ \textbf{in}\ y, (o', o)$$

$$\mathrm{O}[\![\textbf{match}\ V\ \textbf{with inl}\ x :\ M_1\ \mid \textbf{inr}\ y :\ M_2]\!] \triangleq \textbf{match}\ V\ \textbf{with} \mid \textbf{inl}\ x\ \to \textbf{let}\ r, o\ = \mathrm{O}[\![M_1]\!]\ \textbf{in}\ r, \textbf{inl}\ o \mid \textbf{inr}\ y\ \to \textbf{let}\ r, o\ = \mathrm{O}[\![M_2]\!]\ \textbf{in}\ r, \textbf{inr}\ o$$

$$\mathrm{O}[\![M]\!] \triangleq M, () \qquad \text{otherwise}$$

Proposition 4. *If* $\Gamma \vdash M : t$ *then there is* u *such that* $\Gamma \vdash \mathrm{O}[\![M]\!] : t * u$.

Theorem 3. *If* $\Gamma \vdash M : t$ *and* $N = \textbf{let}\ \mathsf{r},\mathsf{o} = \mathrm{O}[\![M]\!]\ \textbf{in observe}\ \mathsf{o};\mathsf{r}$ *and* $\mu \models \mathscr{A}[\![N]\!]_{\mathrm{dom}(\Gamma)}$ *then* $\mathscr{A}[\![M]\!]_{\mathrm{dom}(\Gamma)}\ \mu = \mathscr{A}[\![N]\!]_{\mathrm{dom}(\Gamma)}\ \mu$.

Corollary (Normal form for closed terms)**.** *If* $\diamond \vdash M : t$ *and* N *is as above and* $\delta_{()} \models \mathscr{A}[\![N]\!]_{\diamond}$ *then* $\mathscr{M}[\![M]\!] = \mathscr{M}[\![N]\!]$.

The program $M =$ **observe** (**fst** (hybrid2)) does not satisfy the compatibility precondition of the normal form theorem. This can be checked using static analyses (Bhat et al. 2012a) that can determine if $\mathscr{M}[\![\mathrm{O}[\![M]\!]]\!]$ is absolutely continuous and thus admits a density. A more sensitive analysis could additionally determine the discontinuities of the density function in many cases, allowing to check compatibility statically.

The results of this section imply that **observe**-free models (that is, generative ones), and models with discrete observations only, have measure-transformer semantics that correspond to their monadic semantics, so we can use the latter for its simplicity.

7. Generic Probabilistic Conditionals

Perhaps surprisingly, the lowly **if** expression gives rise to three useful and interesting ways of composing Bayesian models: mixture models, model averaging and a mixture of experts. We also show how to use the **if** expression as a general means for computing model evidence and evidence ratios.

7.1 Bayesian Mixture Models

Given a number of models m_k with $k = 1 \ldots K$ with the same types of inputs x_i and IID data y_i, we can create a mixture of these models by introducing an indicator variable z_i (conditionally independent given w) for each sample y_i that indicates which mixture component m_k it was generated by. This composition is helpful when the data can be generated by one of several known models. Below, we write w_k and h_k for the parameters and the hyperparameter associated with model k, and $\overline{w}$ for $w_1, \ldots, w_K$. The sampling distribution is then given by:

$$p(\{y_i\}_{i=1}^n | \{x_i\}_{i=1}^n, w, \overline{w}) = \prod_{i=1}^{n} \sum_{k=1}^{K} p(z_i = k|w) p(y_i|x_i, m_k, w_k) \quad (6)$$

We here give a generic combinator for creating a mixture of two models, given a prior over the probability of choosing the first distribution.

Mixture Model Combinator

```
let M (m1:Model<'TH1,'TW1,'TX,'TY>,
       m2:Model<'TH2,'TW2,'TX,'TY>)
       : Model<(BetaW * 'TH1 * 'TH2),
                (real * 'TW1 * 'TW2),'TX,'TY> =
  {HyperParameter = (uniformBeta,
                     m1.HyperParameter,m2.HyperParameter)
   Prior = <@ fun (bw,h1,h2) ->
                (random(Beta(bw.trueCount,bw.falseCount)),
                 (%m1.Prior) h1, (%m2.Prior) h2) @>
   Gen= <@ fun ((bias,w1,w2),x) ->
           if breakSymmetry(random(Bernoulli(bias)))
           then (%m1.Gen) (w1,x)
           else (%m2.Gen) (w2,x) @>}
```

(The call to breakSymmetry is a pragma for the Infer.NET backend—it avoids the pitfall of learning symmetric solutions.)

We can create a mixture of two Gaussian models, where we have no prior knowledge of the probability of each model being used.

Mixture of two Gaussian models

```
let m = M(GaussianModel.M,GaussianModel.M)
```

Such a model is useful for fitting a scalar property of two populations of unknown sizes.

7.2 Model Evidence by a Conditional

An important concept in Bayesian machine learning is the notion of *model evidence*, which intuitively is the likelihood of the model given a particular set of observations. It is for instance used to choose between different models of the same data (*model selection* (MacKay 2003, ch.22)), and as an objective function to maximize in certain inference techniques.

If M is a closed term we define $\mathscr{E}(M) \triangleq |\mathscr{M}[\![M]\!]|$, that is, the total measure of the semantics $\mathscr{M}[\![M]\!]$. Note that $\mathscr{E}(M)$ may be different from 1 if M contains instances of **observe** or **fail**, as discussed in Section 6. More generally, if $\Gamma \vdash M : t$ we let $\mathscr{E}(M,\mu) \triangleq |\mathscr{A}[\![M]\!]_{\mathrm{dom}(\Gamma)}\ \mu|$. When choosing between two (or more) different models M and N (for the same data), we typically want to choose the model that has the highest likelihood given the observations (of the data).

The presence of arbitrary **if**-expressions in the language allows to compute the evidence of a model in a uniform way. More generally, the ratio of the evidence of two models (also known as the Bayes factor) can be computed as the ratio between the probabilities of the two possible outcomes of a Boolean variable (compare Minka and Winn (2008)), by inferring the posterior probability of an a priori unbiased selector variable being true.

Evidence ratio by if-expression: $\mathscr{E}[\![M,N]\!]$

```
E[[M,N]] ≜ let x = random(Bernoulli(0.5)) in
           if x then M; () else N; ()
           x
```

Lemma 5. *If* $\Gamma \vdash M : t$ *and* $\Gamma \vdash N : u$ *and* $\mu \in \mathtt{M}\,(\mathrm{range}(\Gamma))$ *then*

$$\frac{\mathscr{E}(M,\mu)}{\mathscr{E}(N,\mu)} = \frac{\mathscr{A}[\![\mathscr{E}[\![M,N]\!]]\!]_{\mathrm{dom}(\Gamma)}\ \mu\ \{(s,\mathbf{true}) \mid s \in \mathbf{V}\mathrm{range}(\Gamma)\}}{\mathscr{A}[\![\mathscr{E}[\![M,N]\!]]\!]_{\mathrm{dom}(\Gamma)}\ \mu\ \{(s,\mathbf{false}) \mid s \in \mathbf{V}\mathrm{range}(\Gamma)\}}.$$

Proof: By expanding $\mathscr{A}[\![\mathscr{E}[\![M,N]\!]]\!]_{\mathrm{dom}(\Gamma)}$. ∎

Given Lemma 5, we can compute the evidence of a model M from its evidence ratio to the trivial model (), which has evidence 1.

Theorem 4. *If* $\diamond \vdash M : t$ *we have* $\mathscr{E}(M) = 2 \cdot \mathscr{M}[\![\mathscr{E}[\![M,()]\!]]\!]\ \{\mathbf{true}\}$. *For open terms, if* $\Gamma \vdash M : t$ *and* $\mu \in \mathtt{M}\,(\mathrm{range}(\Gamma))$ *then* $\mathscr{E}(M,\mu) = 2 \cdot (\mathscr{A}[\![\mathscr{E}[\![M,()]\!]]\!]_{\mathrm{dom}(\Gamma)}\ \mu\ \{(s,\mathbf{true}) \mid s \in \mathbf{V}\mathrm{range}(\Gamma)\})$.

For this way of computing model evidence to work, it is critical that **observe** denotes *unnormalized* conditioning. It does not generalize to languages such as Church (Goodman et al. 2008), where observations (queries) correspond to a renormalized distribution: in such a language $\mathscr{M}[\![\mathscr{E}[\![M,N]\!]]\!]\ \{\mathbf{true}\} = 0.5$ for all admissible M,N. The technique also does not work in languages where the probability distribution of a program is the limit of its successive approximations, such as Probabilistic cc (Gupta et al. 1999). In such a language $\mathscr{M}[\![\mathscr{E}[\![M,()]\!]]\!]\{\mathbf{true}\} \leq 0.5$ for all M. Frameworks based on Gibbs sampling such as BUGS also fail, since **if**-expressions "containing [random] variables require a facility for computing the evidence of a submodel, which Gibbs sampling does not provide" (Minka and Winn 2008).

7.3 Model averaging

In Section 7.1 we were combining models m_k by choosing a model for each data point. Another standard notion of composition is *model averaging* (Hoeting et al. 1999), where we have some prior belief $p(m_k|h)$ about how likely each model m_k is to have generated all of the data, that is then updated based on the evidence of each model given the data, that we here assume to be IID.

$$p(\{y_i\}_{i=1}^n | \{x_i\}_{i=1}^n, w, \overline{w}) = \sum_{k=1}^{K} p(z = k|w) \prod_{i=1}^{n} p(y_i|x_i, m_k, w_k) \quad (7)$$

We show below a combinator for the case $K = 2$. The parameter of the combined model includes a random **bool** switch, which is inspected to generate each output y_i from the model m_{switch}.

Combinator for Model Averaging:

```
let M (m1:Model<'TH1,'TW1,'TX,'TY>,
       m2:Model<'TH2,'TW2,'TX,'TY>)
    : Model<(real*'TH1*'TH2),
             (bool*'TW1*'TW2),'TX,'TY> =
 {HyperParameter = (0.5,
                    m1.HyperParameter,m2.HyperParameter)
  Prior = <@ fun (bias,h1,h2) →
              (random(Bernoulli(bias)),
               (%m1.Prior) h1, (%m2.Prior) h2) @>
  Gen = <@ fun ((switch,w1,w2),x) →
           if switch then (%m1.Gen) (w1,x)
           else (%m2.Gen) (w2,x) @>}
```

The posterior distribution over the bias gives us the evidence ratio between the two models m1 and m2 (cf. Lemma 5), multiplied with the prior odds.

$$\frac{p(z=1|d,h)}{p(z=2|d,h)} = \frac{p(d|m_1,h_1)}{p(d|m_2,h_2)} \frac{p(z=1|h)}{p(z=2|h)} \tag{8}$$

7.4 Mixture of Experts

A powerful supervised counterpart to the unsupervised mixture models discussed in 7.1 is the Mixture of Experts model (Jacobs et al. 1991), and its hierarchical variants (Bishop and Svensén 2003; Jordan and Jacobs 1994). The idea is to use a so-called gating model $p(z_i|x_i,w)$ to decide for each input x_i which model to use for generating the corresponding output y_i. We consider a mixture of K models with conditional sampling distributions $p(y_i|x_i,m_k,w_k)$, priors $p(w_k|h_k)$ and a gating model $p(z_i|x_i,w)$ with prior $p(w|h)$ resulting in the combined sampling distribution

$$p(\{y_i\}_{i=1}^n|\{x_i\}_{i=1}^n,w,\overline{w}) = \prod_{i=1}^{n}\sum_{k=1}^{K} p(z=k|x_i,w)p(y_i|x_i,m_k,w_k) \tag{9}$$

As before, we here implement the binary case with two data models m_k with $k \in \{T,F\}$, with independent priors given their parameters: $p(w,w_F,w_T|h,h_T,h_F) = p(w|h)p(w_T|h_T)p(w_F|h_F)$.

Combinator for Mixture of Experts:

```
let M(mc:Model<'TH,'TW,'TX,bool>,
      m1:Model<'TH1,'TW1,'TX,'TY>,
      m2:Model<'TH2,'TW2,'TX,'TY>)
    : Model<'TH*'TH1*'TH2,'TW*'TW1*'TW2,'TX,'TY> =
 {HyperParameter = (mc.HyperParameter,
                    m1.HyperParameter,m2.HyperParameter)
  Prior = <@ fun (hc,h1,h2) →
      (%mc.Prior) hc, (%m1.Prior) h1, (%m2.Prior) h2 @>
  Gen = <@ fun ((wc,w1,w2),x) →
           if (%mc.Gen) (wc,x) then (%m1.Gen) (w1,x)
           else (%m2.Gen) (w2,x) @>}
```

The hierarchical Mixture of Experts model (Bishop and Svensén 2003; Jordan and Jacobs 1994) can easily be obtained by a tree of calls to the above combinator. It is also straightforward to build an n-ary version of this combinator from the above construction, or a variation that operates on arrays of models with identical type parameters.

Some Examples of Hierarchical Mixtures of Experts

```
let m = ExpertMixture.M(Probit,M2,M2)
let n = IIDArray.M(ExpertMixture.M(Probit,m,m))
```

The model M2 fits a line to the data. The model m above attempts to fit two different lines l_1 and l_2, and to find a point x_0 where the data gradually shifts from being fitted to l_1 (for $x \ll x_0$) and being fitted to l_2 (for $x \gg x_0$). The model n attempts to fit four lines with three separating points, and operates on arrays of IID data.

8. A Learner based on Monte Carlo Inference

Markov chain Monte Carlo (MCMC) methods are an important class of inference algorithms for Bayesian models because they make it possible to obtain samples from the posterior for a wide variety of models, even if that posterior is not in a simple family of parametric probability distributions or densities. 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) provides an excellent review of MCMC methods. As an example implementation, we consider Filzbach (Purves and Lyutsarev 2012), an implementation of an adaptive MCMC sampler based on the Metropolis-Hastings algorithm (Hastings 1970; Metropolis et al. 1953). All that is required to apply an MCMC algorithm to a particular model is the ability to calculate posterior probabilities for a given set of parameters. The algorithm then generates an ensemble of samples from the posterior. This ensemble serves as a representation of the posterior, or to calculate desired marginal distributions of individual parameters or other integrals under the posterior distribution.

8.1 Direct Calculation of Log-Posterior

Bhat et al. (2012a) report an algorithm (but not an implementation), for computing the densities of the distribution computed by a probabilistic program, and a type system that guarantees their existence. Their algorithm uses integration at every **let**-expression; we here implement a simplified version without integrals but instead limited to deterministic **let**-bound variables only. This pattern is common in the real-world models we have studied, for greater efficiency and simplicity. (In unpublished work, Bhat et al. (2012b) implement a more general algorithm for calculating densities of Fun programs.)

We describe a recursive function PRE, which is an effective partial algorithm for calculating the log-density for the parameters of a generative model written in Fun. (It is standard to compute logs to avoid underflow when dealing with very small probabilities.) The PRE function is a backward analysis that computes the density function, and is named by analogy with the standard pre-condition computation in program analysis.

Log-density calculation $\text{PRE}(V,\sigma,M) \in \mathbb{R} \cup \{-\infty, \bot\}$

$$\text{PRE}(V,\sigma,U) = \log(1.0) \textbf{ if } U\sigma = V \textbf{ else } \log(0.0)$$

$$\text{PRE}(V,\sigma,f(V_1,\ldots,V_n)) = \log(1.0) \textbf{ if } f(V_1\sigma,\ldots,V_n\sigma) = V \textbf{ else } \log(0.0) \textbf{ otherwise}$$

$$\text{PRE}(V,\sigma,\textbf{random}(\text{Dist}(U))) = \log(D(\mu_{\text{Dist}(U\sigma)}(V)))$$

$$\text{PRE}(V,\sigma,\textbf{match } U \textbf{ with inl } x: M \mid \textbf{inr } y: N) = \text{PRE}(V,(\sigma,x\mapsto U'),M) \textbf{ if } U\sigma = \textbf{inl } U' \text{ PRE}(V,(\sigma,y\mapsto U'),N) \textbf{ if } U\sigma = \textbf{inr } U'$$

$$\text{PRE}(V,\sigma,\textbf{let } x = M \textbf{ in } N) = \text{PRE}(V,(\sigma,x\mapsto U),N) \textbf{ if } M \text{ is deterministic and } M\sigma \text{ evaluates to } U \bot \textbf{ otherwise}$$

$$\text{PRE}(V,\sigma,\textbf{fail}) = \log(0.0)$$

$$\text{PRE}(V,\sigma,\textbf{observe } f(\overline{V})) = \bot$$

When defined (that is, not $\bot$), the $\text{PRE}(V,\sigma,M)$ function computes the log-posterior density of the parameters (in σ) given the data V.

Theorem 5. *If* $\Gamma \vdash M : t$ *and* σ *is a* Γ*-valuation and* $V \in \mathbf{V}_t$ *and* $\text{PRE}(V,\sigma,M) \neq \bot$ *then* $\text{PRE}(V,\sigma,M) = \log((D(\mathscr{P}[\![M]\!]\ \sigma))(V))$.

The log-posterior of the parameters V_w is then obtained by adding the log-density of the prior at V_w, which is obtained by calling $\text{PRE}(V_w, h \mapsto V_h, \text{Prior})$, where V_h is the hyperparameter, to the log-posterior density $\text{PRE}(V_y, w \mapsto V_w, M)$.

Here is the actual log-posterior function we pass to Filzbach from our MCMC learner (space precludes more detail):

```
let (Lambda(vh,p)) = m.Prior
let (Lambda(vw,e)) =
    <@ fun w → [|for x in xs → (%m.Gen) (w,x)|] @>
let likelihood : 'TW → real =
    (* fun w → PRE(ys,[vw,w],e) + PRE(w,[vh,h],p) *)
    Compile(fun w → <@ (%PRE(<@ ys @>,[vw,w],e)) +
                       (%PRE(w,[vh,<@ h @>],p)) @>)
```

Here, (xs,ys) is the observed data (from training) and m is the original model. Though not shown here, the MCMC learner represents distributions simply as an ensemble (an array) of samples.

8.2 Example: Purves' PlantGrowth

As a typical example, we present a simplified model of *plant growth*, adapted from a Filzbach example. Given an array of (deterministic) temperature data, temps, the model generates the mass of a plant on harvest day x, assuming that $0 \leq x < |\text{temps}|$.

The parameter of the model is a record of numbers: the plant's initial mass (imass), daily growth rate (alpha), optimum temperature for growth (topt), temperature sensitivity (trho) and the deviation of some noise (sigma). Each field is given a Gaussian prior.

The harvest mass is calculated by iterating a daily growth function (the argument of Array.Fold below) on the initial mass. This deterministic computation is finally perturbed by some noise, drawn from a Gaussian of unknown precision (prec obtained from sigma).

PlantGrowth Model (Prior **omitted due to lack of space) :**

```
<@ fun (w,x) → // first grow plant deterministically...
    let prec = 1.0/(w.sigma * w.sigma)
    let mass = // compute the mass after x days
        Array.fold // compute next mass in a time series
            (fun mass_i airtemp_i →
              let r = (airtemp_i−w.topt)/w.trho
              mass_i + (w.alpha * mass_i) * exp (−1.0*(r*r)))
            w.imass (Array.sub temps 0 x)
    random(Gaussian(mass,prec)) @> // ... finally add noise
```

To perform inference on this model, a Filzbach user must also provide a function computing the log-posterior of the model's parameters given some observed data (here, pairs of harvest days and masses). It is this function that Filzbach's MCMC algorithm attempts to maximize over the parameter space. Moreover, the user typically also needs to write an explicit generative model for producing predictions, based either on the mean of the obtained posterior parameter distribution or, better yet, by *error propagation* using an ensemble of parameters drawn from this posterior. Instead, we use the PRE algorithm described above to automatically compute the log-posterior function from the generative model.

(Adaptation of our learner to related methods (such as the computation of either the *maximum likelihood* (MLE) or the *maximum a posteriori probability* (MAP) estimate) appears straightforward but is beyond the scope of the paper.)

9. Related Work

Formal Semantics of Probabilistic Languages There is a long history of formal semantics for probabilistic languages with sampling primitives, often with recursive computation. One of the first semantics is for Probabilistic LCF (Saheb-Djahromi 1978), which augments the core functional language LCF with weighted binary choice, for discrete distributions. Kozen (1981) develops a probabilistic semantics for while-programs with random assignment. He develops two equivalent semantics; one more operational, and the other denotational. McIver and Morgan (2005) develop a theory of abstraction and refinement for probabilistic while programs with real-valued variables, based on weakest preconditions.

Jones and Plotkin (1989) investigate the probability monad, and apply it to languages with discrete probabilistic choice. Ramsey and Pfeffer (2002) give a stochastic λ-calculus with a measure-theoretic semantics in the probability monad, and provide an embedding within Haskell; they also do not consider observations.

Several languages (such as PFP (Erwig and Kollmansberger 2006)), IBAL (Pfeffer 2007), HANSEI (Kiselyov and Shan 2009), CertiPriv (Barthe et al. 2012) and the informal sampling semantics of Csoft (Minka et al. 2009)) have Boolean assertions or **fail** statements that give rise to sub-probabilities. Such languages support the mixture operations and evidence calculations of Section 7, for discrete observations only.

The concurrent constraint programming language PCC of Gupta et al. (1999) allows describing continuous probability distributions using independent sampling and constraints. However, their semantics of constraints is different than that of Fun observations. This makes PCC less suitable for compositional Bayesian modelling, since computation of model evidence using an if statement (Theorem 4) does not work: the program `choose` X `from` $\{0,1\}$ `in if` $X = 1$ `then` $[b,P]$ yields probability at most 0.5 of b being true.

In contrast, the measure-transformer semantics of Fun allows computing $\mathscr{E}(M) > 1$ for well-fitted models with continuous observations using the one-sided if statement of Theorem 4, as do the direct factor graph semantics of Csoft (Minka et al. 2009) using gates (Minka and Winn 2008).

Probabilistic Languages for Machine Learning Previous programming models for machine learning mainly represent particular implementation strategies, rather than general Bayesian models.

An exception is Church (Goodman et al. 2008), which represents probability distributions over Lisp terms as Lisp programs, and uses various inference techniques. However, since every admissible term denotes a probability distribution, $\mathscr{E}[\![M, ()]\!]$ always yields true with probability 0.5.

Early work includes that of Koller et al. (1997) where a probabilistic model is represented as a functional program with probabilistic choice. FACTORIE (McCallum et al. 2009) is a library for imperatively constructing factor graphs. BUGS (Gilks et al. 1994) uses Gibbs sampling for Bayesian inference on Bayesian networks, represented as imperative probabilistic programs. IBAL (Pfeffer 2007) integrates Bayesian parameter estimation and decision-theoretic utility maximisation, but only works with discrete datatypes. Alchemy (Domingos et al. 2008) represents Markov logic networks, which are inherently discrete. Park et al. (2005) also define a probabilistic functional language, where as in Church terms always denote probability distributions.

PMML (Guazzelli et al. 2009) is an XML format for representing trained instances of a range of models, including association rules, neural networks, decision trees, support vector machines, etc. PMML does not represent arbitrary graphical or generative models. In our terms, a PMML model corresponds roughly to a trained learner. We have not developed any format for persisting trained learners; a format based on PMML may be suitable.

Example / Learner	TH	TW	TDistW	TX	TY	TDistY
Sprinkler / A	SP.TH	SP.TW<**bool**>	ADD<SP.TW<**bool**>>	SP.TX	**bool**	ADD<**bool**>
TwoCoins / A	TC.TH	TC.TW<**bool**>	ADD<TC.TW<**bool**>>	TC.TX	**bool**	ADD<**bool**>
Two Coins / IN	TC.TH	TC.TW<**bool**>	TC.TW<Bernoulli>	TC.TX	**bool**	Bernoulli
Friends / A	**bool**[][]	**bool** list list	ADD<**bool** list list>	**int** ∗ **int** ∗ **int**	**bool**	ADD<**bool**>
Students / A	**int** ∗ **int**	**bool** list list	ADD<**bool** list list>	**int** ∗ **int** ∗ **int**	**bool**	ADD<**bool**>
Gaussian / IN	GM.TH	GM.TW<$\mathscr{R}$,$\mathscr{R}$>	GM.TW<$\mathscr{N}$,Γ>	**unit**	**real**	$\mathscr{N}$
Gaussian Mix/ IN	MX1.TH	$\mathscr{R}$∗ GaussW∗GaussW	β∗ GM.TW<$\mathscr{N}$,Γ>∗GM.TW<$\mathscr{N}$,Γ>	**unit**	**real**	$\mathscr{N}$
Gaussian Mix / F	MX2.TH	(GaussW∗GaussW)	(GaussW∗GaussW)[]	**unit**	**real**	$\mathscr{R}$[]
PlantGrowth / F	**unit**	PG.TW	PG.TW[]	**int**	**real**	$\mathscr{R}$[]
TrueSkill / IN	TS.TH	TS.TW<$\mathscr{R}$>	TS.TW<$\mathscr{N}$>	TrueSkill.TX	**bool**	Bernoulli
Lin. Reg. / IN	LR.TH	LR.TW<$\mathscr{R}$,$\mathscr{R}$,$\mathscr{R}$>	LR.TW<$\mathscr{N}$,$\mathscr{N}$,Γ>	**real**	**real**	$\mathscr{N}$
MV Gaussian / IN	MVG.TH	MVG.TW<$\vec{\mathscr{R}}$,$\mathscr{M}$>	MVG.TW<$\vec{\mathscr{N}}$,$\mathscr{W}$>	**unit**	$\vec{\mathscr{R}}$	$\vec{\mathscr{N}}$

$\mathscr{R}$= **real** $\mathscr{N}$ = Gaussian β= Beta Γ= Gamma
$\vec{\mathscr{R}}$= Vector $\mathscr{M}$= PositiveDefiniteMatrix
$\mathscr{W}$ = Wishart // *generalizes* Γ *to multiple dimensions*
$\vec{\mathscr{N}}$ = VectorGaussian // *multivariate Gaussian distribution*
GaussW= {Mean:$\mathscr{R}$; Precision:$\mathscr{R}$}
BetaW = {trueCount: $\mathscr{R}$; falseCount: $\mathscr{R}$}
SP.TH = {RainH: $\mathscr{R}$; SprinklerH: $\mathscr{R}$}
SP.TW<'TB> = {Rain: 'TB; Sprinkler: 'TB}
SP.TX = IsGrassWet // *a unit type*
TC.TH = {Bias1: $\mathscr{R}$; Bias2: $\mathscr{R}$}
TC.TW<'TB> = {Heads1: 'TB; Heads2: 'TB}
TC.TX = AreEitherHeads // *a unit type*
GM.TW<'TM,'TP> = {Mean:'TM; Precision:'TP}
GM.TH ={ Gaussian: GaussW, Gamma: GammaW }
MX1.TH = BetaW ∗ GM.TH ∗ GM.TH
MX2.TH = GM.TH ∗ GM.TH
PG.TW = {alpha:$\mathscr{R}$;topt:$\mathscr{R}$;trho:$\mathscr{R}$;imass:$\mathscr{R}$;sigma:$\mathscr{R}$}
TS.TH = { Players: **int**; G: GaussW; P: GammaW }
TS.TW<'TA> = { Skills: 'TA[]}
TS.TX = { P1:**int**; P2: **int** }
LR.TH = {MeanA: $\mathscr{R}$; PrecA: $\mathscr{R}$; MeanB: $\mathscr{R}$; PrecB: $\mathscr{R}$;
Shape: $\mathscr{R}$; Scale: $\mathscr{R}$}
LR.TW<'TA,'TB,'TN> = {A:'TA; B:'TB; Prec:'TN}
MVG.TH = {NCols:**int**; MeanVectorPrecisionCount:$\mathscr{R}$;
WishartShapeConstant:$\mathscr{R}$; WishartScaleConstant:$\mathscr{R}$}
MVG.TW<'TM,'TC> = {Mean:'TM; Covariance:'TC}

Table 1. Rows show types for L : ILearner(TDistW,TX,TY,TDistY) for m : Model<TH,TW,TX,TY> (A=ADD, IN=Infer.NET, F=Filzbach)

Inference using ADDs The idea of using ADDs for probabilistic inference has been explored before. Sanner and McAllester (2005) define Affine Algebraic Decision Diagrams to perform inference over Bayesian networks and Markov Decision Processes. Kwiatkowska et al. (2006) have used a variants of ADDs to perform probabilistic model checking in the PRISM project. Bozga and Maler (1999) have used ADDs to symbolically simulate Markov chains. Chavira and Darwiche (2007) use ADDs to compactly represent factors in a Bayesian network and thereby perform efficient inference via variable elimination. In contrast, the ADD backend described in this paper avoids factor graphs altogether and uses ADDs to represent symbolic program states (which are distributions) at every program point, much like a data-flow analysis or an abstract interpreter (Cousot and Cousot 1977).

Mardziel et al. (2011) develop an approximate representation for discrete distributions based on abstract interpretations, with application to knowledge-based security policies.

10. Conclusions

We proposed typed programming abstractions, models and learners, for packaging, composing, and training and predicting with Bayesian models. We are aware of no prior generic abstractions for Bayesian models.

Future work includes adding functions over recursive types, and general recursion to the Fun language. We will also investigate the possibility of relaxing the condition of compatibility of a measure with respect to a model.

Acknowledgements Conversations about this work with Chris Bishop, John Bronskill, Tom Minka, Drew Purves, Robert Simmons, Matthew Smith, and John Winn were invaluable. Sooraj Bhat, Jael Kriener, and Sameer Singh commented on a draft.

References

G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In J. Field and M. Hicks, editors, *POPL*, pages 97–110. ACM, 2012.

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, 2012a.

S. Bhat, J. Borgström, A. D. Gordon, and C. Russo. Deriving probability density functions from probabilistic functional programs. Draft paper, 2012b.

C. M. Bishop and M. Svensén. Bayesian hierarchical mixtures of experts. In C. Meek and U. Kjärulff, editors, *Uncertainty in Artificial Intelligence (UAI'03)*, pages 57–64. Morgan Kaufmann, 2003.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

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`.

M. Bozga and O. Maler. On the representation of probabilities over structured domains. In *Computer Aided Verification (CAV '99)*, pages 261–273, 1999.

M. Chavira and A. Darwiche. Compiling Bayesian networks using variable elimination. In *International Joint Conference on on Artificial Intelligence (IJCAI '07)*, pages 2443–2449, 2007.

G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference for probabilistic programs via symbolic execution. Technical Report MSR–TR–2012–86, Microsoft Research, 2012.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. CUP, 2009.

H. Daumé III. *HBC: Hierarchical Bayes Compiler*, 2008. Available at `http://www.cs.utah.edu/~hal/HBC/`.

P. Domingos, S. Kok, D. Lowd, H. Poon, M. Richardson, and P. Singla. Markov logic. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic inductive logic programming*, pages 92–117. Springer-Verlag, Berlin, Heidelberg, 2008.

M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16(1):21–34, 2006.

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.

N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI'08)*, pages 220–229. AUAI Press, 2008.

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. Technical Report MSR-TR-2013-1, Microsoft Research, 2013.

A. Guazzelli, M. Zeller, W. Chen, and G. Williams. PMML: An open standard for sharing models. *The R Journal*, 1(1), May 2009.

V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *POPL*, pages 189–202, 1999.

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

R. Herbrich, T. Minka, and T. Graepel. Trueskill$^{\text{tm}}$: A Bayesian skill rating system. In *Advances in Neural Information Processing Systems (NIPS'06)*, pages 569–576, 2006.

J. A. Hoeting, D. Madigan, A. E. Raftery, and C. T. Volinsky. Bayesian model averaging: A tutorial. *Statistical Science*, 14(4):382–401, 1999.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science (LICS'89)*, pages 186–195. IEEE Computer Society, 1989.

M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence (UAI'09)*, 2009.

D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.

D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

M. Z. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. In *Quantitative Aspects of Programming Languages (QAPL 2005)*, volume 153(2) of *ENTCS*, pages 5–31, 2006.

D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. CUP, 2003.

P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Computer Security Foundations Symposium (CSF'11)*, pages 114–128, 2011.

A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems (NIPS'09)*, pages 1249–1257, 2009.

A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems*. Monographs in computer science. Springer, 2005.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

T. Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, MIT, 2001.

T. Minka and J. M. Winn. Gates. In *Advances in Neural Information Processing Systems (NIPS'08)*, pages 1073–1080. MIT Press, 2008.

T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from `http://research.microsoft.com/infernet`.

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.

S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL*, pages 171–182. ACM, 2005.

J. Pearl and G. Shafer. Probabilistic reasoning in intelligent systems: Networks of plausible inference. *Synthese-Dordrecht*, 104(1):161, 1995.

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

A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 733–740. Morgan Kaufmann, 2001.

A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

A. Pfeffer. Practical probabilistic programming. In P. Frasconi and F. A. Lisi, editors, *Inductive Logic Programming (ILP 2010)*, volume 6489 of *Lecture Notes in Computer Science*, pages 2–3. Springer, 2010.

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`.

A. Radul. Report on the probabilistic language scheme. In *Proceedings of the 2007 symposium on Dynamic languages (DLS'07)*, pages 2–10. ACM, 2007.

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

N. Saheb-Djahromi. Probabilistic LCF. In *Mathematical Foundations of Computer Science (MFCS)*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.

S. Sanner and D. A. McAllester. Affine Algebraic Decision Diagrams (AADDs) and their application to structured probabilistic inference. In *International Joint Conference on on Artificial Intelligence (IJCAI '05)*, pages 1384–1390, 2005.

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.

F. Somenzi. CUDD: CU decision diagram package, release 2.5.0, 2012. Software available from `http://vlsi.colorado.edu`.

D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In A. Kennedy and F. Pottier, editors, *ML Workshop*, pages 43–54. ACM, 2006.

J. Winn and T. Minka. Probabilistic programming with Infer.NET. Machine Learning Summer School lecture notes, available at `http://research.microsoft.com/~minka/papers/mlss2009/`, 2009.