A Model-Learner Pattern for Bayesian Reasoning

Andrew D. Gordon (Microsoft Research and University of Edinburgh)  Mihailing 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.

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 (Gils 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), FP (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 \( \text{Model<TH,TW,TX,TY>} \). 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 \( h : \text{TH} \), parameters \( w : \text{TW} \), inputs \( x : \text{TX} \), and outputs \( y : \text{TY} \). 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 penalty. 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. 2015).

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 \( \text{ty}(c) = t \) to mean that constant \( c \) has type \( t \).

Values and Types of Fun:

\[
\begin{align*}
U, V & := x \mid c \mid (V, V) \mid \text{inl} V \mid \text{inr} V & \text{value} \\
a, b & := \text{int} \mid \text{real} & \text{base type} \\
t, u & := \text{unit} \mid (t_1 + t_2) \mid (t_1 + t_2) & \text{compound type}
\end{align*}
\]

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

Distributions:

\[
\text{Dist} : (x_1 : t_1 \times \ldots \times x_n : t_n) \rightarrow \text{PDist}(t)
\]

Bernoulli : (bias : \text{real}) \rightarrow \text{PDist}(\text{bool})

Beta : (a : \text{real} \times b : \text{real}) \rightarrow \text{PDist}(\text{real})

Gaussian : (mean : \text{real} \times \text{prec} : \text{real}) \rightarrow \text{PDist}(\text{real})

Gamma : (shape : \text{real} \times \text{scale} : \text{real}) \rightarrow \text{PDist}(\text{real})

A Bernoulli distribution is a biased coin flip; the bias lies in the unit interval \([0, 1]\) and is the probability of \text{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 \( a = 1 \) and \( b = 1 \), it is the uniform distribution on the unit interval. A Gaussian distribution is parameterized by its mean and precision; the standard deviation \( \sigma \) follows from the identity \( \sigma^2 = 1 / \text{prec} \). 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:

\[
\begin{align*}
M, N & := \text{expression} \\
V & := \text{value} \\
f(V_1, \ldots, V_n) & \text{deterministic application} \\
\text{let } x = M \text{ in } N & \text{let (scope of } x \text{ is } N) \\
\text{match } V \text{ with } \text{inl } x : M & \text{matching (scope of } x \text{ is } M, \text{ inr } y : N \text{ scope of } y \text{ is } N) \\
\text{random(Dist}(V)) & \text{primitive distribution} \\
\text{observe } f(V_1, \ldots, V_n) & \text{observation} \\
\text{fail} & \text{failure}
\end{align*}
\]

We rely on several standard syntactic conventions. We write \( \text{if } V \text{ then } M \text{ else } N \) for \( \text{match } V \text{ with } \text{inl } x : M \mid \text{inr } x : N \). We write \( M; N \) for \( \text{let } x = M \text{ in } N \) 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 \( \text{observe } f(V) \), where \( V \) is short for \( V_1, \ldots, V_n \). Such an observation expresses (unnormalized) conditioning on the event that the indicator function \( f(V) \) yields a zero, defined as a closed value where every instance of \( c \) is 0. The primitive \text{fail} represents an impossible event. It has the same meaning as an observation such as \( \text{observe } (1 + 1) \) whose indicator function cannot yield zero. Note that \text{observe} is a no-op on \text{bool}, since any Boolean is a zero according to the above definition. We write \( \text{observe } (x = V) \) for \( \text{observe } (x - V) \) 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 \( \Gamma = \{ x_1 : t_1, \ldots, x_n : t_n \} \) the expression \( M \) has type \( t \). We write \( \circ \) for the empty type environment. Apart from the following, the typing rules are standard.

Selected Typing Rules:

\[
\begin{align*}
\text{(Fun Observe)} & : \Gamma \vdash f(V) : \text{unit} \\
\text{(Fun Random)} & : \Gamma \vdash \text{Dist}(V) : \text{PDist}(t)
\end{align*}
\]

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 \( \text{Expr<'}\text{>}. \) More generally, if \( \circ, x_1 : t_1, \ldots, x_n : t_n \vdash M : t \) we represent \( M \) by the F# quotation \( <\!\!\circ \text{ fun}(x_1, \ldots, x_n) \rightarrow M\!\!> \). Suppose \( E \) is a quotation; \( \text{Eval} \) 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 \( \theta[M] \) 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 \( \theta[M] \). We can represent an inference engine as a function \( \text{Expr<'}\text{> \rightarrow Dist}\text{T} \) where \( \text{Dist}\text{T} \) 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 known as the coordinates. Intuitively, we fit a noisy line $y_j = Ax_j + B + \varepsilon$ to the data, where the noise $\varepsilon$ is drawn from a Gaussian distribution with mean $0$ and precision $\text{Prec}$.

The expression $\text{prior } h$ below expresses our initial uncertainty about the parameters $A, B$, and $\text{Prec}$ 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 $\text{gen}(w,x)$ defines how to generate each $y_i$ from $x_i$ and parameters $w_i$. (In the Fun code below, we use record and array notations which are eventually treated as operations on tuples.)

The Prior and Sampling Distributions:

\begin{verbatim}
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, 0.0, Prec))]} //These expressions formalize the prior and the data sampling. The prior distribution $p(w|h)$ is the density of $\mathcal{M}[\text{prior } h]$ at $w$, while the sampling distribution $p(y|x,w)$ is the density of $\mathcal{M}[\text{gen}(w,x)]$ 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 $\mathcal{M}[\text{predictive}(h,x)]$ at $y$. The Prior Predictive Distribution:

\begin{verbatim}
let predictive (h,x) = let w = prior h in gen(w,x)
\end{verbatim}

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

\begin{equation}
p(y|x,h) = \int p(y|x,w)p(w|h)dw
\end{equation}

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

\begin{verbatim}
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]
\end{verbatim}

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

\begin{verbatim}
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)
\end{verbatim}

We express the posterior and posterior predictive distributions as Fun expressions, using \text{observe} to condition on the data $(x,y)$.

The Posterior and Posterior Predictive Distributions:

\begin{verbatim}
let posterior (h,x,y) = let w = prior h in observe(y=gen(w,x)); w
let posterior predictive(h,x,y') =
    let w = posterior (h,x,y) in gen(w,x')
\end{verbatim}

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

\begin{equation}
p(w|d,h) = \frac{p(y|x,w)p(w|h)}{p(d|h)}
\end{equation}

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:

\begin{equation}
p(y'|x',d,h) = \int p(y'|x',w)p(w|d,h)dw
\end{equation}

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.

\begin{verbatim}
{ A = Gaussian(0.5576, 0.05089); // actual A = 0.70 B = Gaussian(4.936, 0.404); // actual B = 4.33 Prec = Gamma(1.695, 0.460|mean=0.78); // actual Prec=0.58
\end{verbatim}

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') we obtain the following.

\begin{verbatim}
[ Gaussian(8.282, 0.1888); Gaussian(8.839, 0.1654); Gaussian(9.397, 0.1469); Gaussian(9.954, 0.1303); ]
\end{verbatim}

To summarize, we modelled a noisy line by distributions written as the Fun expressions $\text{prior } h$ and $\text{gen}(w,x)$. 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<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:**

\begin{verbatim}
type Model<TH, TW, TX, TY> = { HyperParameter: TH; Prior: Expr<TH -> TX>; Gen: Expr<TW * TX -> TY > }
\end{verbatim}

Given a model $m$, the Fun expression $h = m$.HyperParameter is the hyperparameter, the Fun expression $(\%m.Prior)$ 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 $(\%m.Prior)$ 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 $<@\ldots>@$ to treat the bodies of prior and gen above as Fun expressions.
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:

```plaintext
module LinearRegression =
    type TH = {M: A; P: Prec:A; : ... }
    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>
```

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:

```plaintext
module LinearRegression =
    type TH = {M: A; P: Prec:A; : ... }
    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>
```

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 an exact evaluation of our API, by assembling suitable quotations. The mutable variable $d$ takes as values Fun quotations 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.)

```plaintext
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.

```plaintext
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:

$$p(y_1^n | x_1^n, w) = \prod_{i=1}^n p(y_i | x_i, w)$$

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 $x$, as described in Section 2.)

```plaintext
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(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 = < Φ fun h > |\ h Φ > // as before
    Gen = < Φ fun(w,x) →
        (w.A + x) \+ w.B + random(Gaussian(0.0,w,Prec)) Φ >
  :Model<TH,TW<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 = InferNet.Learner.LearnerFromModel(m,Marginalize)
let (w,dW) = testMany toL M2 [[ -100.0 , 1.0 .. 100.0 ]]
```

On one test run, we obtained:

\[
w = \{ \ A = 0.3477837663; \ B = -27.13593566; \ Prec = 2.103660906; \}
dW =\{ \ A = Gaussian(0.3479, 1.467e06); \ B = Gaussian(-27.09, 435.0); \ Prec = Gamma(99.74, 0.02203)[mean=2.197]; \}
\]

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 \( \text{bool} = \text{unit} + \text{unit} \), the only matches are conditionals, there are no instances of \text{observe}, and every \text{random} expression takes the form \text{random} (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 \text{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 \( \text{let } x: = N \in M \) for the corresponding untyped \text{let}-expression when \( \Gamma \vdash N: t \). We write \text{ite}(v = U, V_f, V_t)\( )\) for the conditional statement that returns \( V_f \) when \( v = U \) and \( V_t \) otherwise. A valuation \( \sigma \) is a finite map from variables to closed values; \( \Gamma \)-valuation \( \sigma \) is a finite map \( x_1 \mapsto V_1, \ldots, x_n \mapsto V_n \) where each \( V_i \in V \). We let \( = \emptyset \) denote the empty valuation, and write \( \Sigma(\Gamma) \) for the set of \( \Gamma \)-valuations. If \( \Gamma \vdash V: t \) and \( \sigma \) is a \( \Gamma \)-valuation, \( V\sigma \) is the closed value resulting from substituting \( \sigma(x) \) for \( x \) in \( V \) for all \( x \in \text{dom}(\Gamma) \). Given an open expression \( M \), where \( \Gamma \vdash M: t \), \text{Post} is a function from valuations \( \sigma \in \Sigma(\Gamma) \) to discrete sub-probability distributions over \( V \). By uncurrying, the semantics of \( M \) can also be expressed as a function from \( \Sigma(\Gamma) \times V \), to sub-probabilities \( [0,1] \). The algorithm \text{Post}(\( M \)) computes this function by structural recursion on \( M \). The \text{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** \( \text{Post}(M): \Sigma(\Gamma) \times V \rightarrow \text{real} \) when \( \Gamma \vdash M: t \)

\[
\begin{align*}
\text{Post}(V) &= \lambda \sigma. v. \text{ite}(v = V \sigma, 1, 0) \\
\text{Post}(f(V_1, \ldots, V_n)) &= \lambda \sigma. v. \text{ite}(v = f(V_1 \sigma, \ldots, V_n \sigma), 1, 0) \\
\text{Post}(\text{Bernoulli}(r)) &= \lambda \sigma. v. \text{ite}(v = \text{true}, r, 1 - r) \\
\text{Post}(\text{if}(V \text{ then } N_1 \text{ else } N_2)) &= \\
\lambda \sigma. v. \text{Post}(V)[\sigma, \text{true}] \cdot \text{Post}(N_1)[\sigma, v] + \\
\text{Post}(V)[\sigma, \text{false}] \cdot \text{Post}(N_2)[\sigma, v] \\
\text{Post}(\text{let } x: = N_i \text{ in } N_2) &= \lambda \sigma. v. \Sigma_{u \in V_1} \text{Post}(N_1)[\sigma, u] \cdot \text{Post}(N_2)[\sigma, (x \mapsto u), v] \\
\text{Post}(\text{fail}) &= \lambda \sigma. (1)\].
\end{align*}
\]

The \text{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 \text{if}, operation (3) is used in the cases for values, deterministic functions, and Bernoulli, and operation (4) is used in the \text{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 \( \vdash M: t \). For all values \( V \in V_1 \), \( \text{Post}(M)[\sigma, V] = \mathcal{M}[M][V] \).

The ADD learner builds on the reference learner, calling the \text{Post} algorithm on the Fun expression returned from its \text{Posterior} and \text{Predict} methods. An ADD learner has the following type, where ADD<’T,‘W> is the type of a decision diagram that maps values of type ’T to probabilities.

\[
: \text{ILearner<ADD<’TW>,’TX,’TY,ADD<’TY>>}
\]

In our experiments, the performance of the ADD learner is competitive with algorithms implemented in Samlaml (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 sprinkler, and the view that rain has a
Sprinkler model

```scala
module Sprinkler =
  type TH = {RainH: real; SprinklerH: real}
  type TW = {Rain: 'b; Sprinkler: 'b}
  type TX = {IsGrassWet: bool; SprinklerH: real}
  type TY = bool

let M: Model<TH,TW,TX,TY> =
  let HyperParameter = {RainH=0.3; SprinklerH=0.5}
  let Sprinkler = random(Bernoulli(h.RainH))
  let Gen = <@ fun (w,x) ->
    let random = Bernoulli(0.9) && w.Rain ||
    let 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.

[(true: Rain = false, Sprinkler = false); 0.05777484318);
[(true: Rain = true, Sprinkler = false); 0.2253218884];
[(false: Rain = true, Sprinkler = true); 0.4737537141];
[(true: 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 use 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 `d1` say of the distribution type `Bernoulli`, and computes the approximate marginal of the second projection as a value `d2` 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:

```scala
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`.

Constrcuting an Infer.NET Learner:

```scala
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:

```scala
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=0.0;Precision=1.0}
    Gamma=Shape1.1:Scale2.0}
  let p = h.Gaussian.Precision
  let m = h.Gaussian.Mean
  let sh = h.Gamma.Shape
  let sc = h.Gamma.Scale

Gen = <@ fun (w,x) ->
  let m,p = 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).

```scala
: 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:

```latex
let Probit : Model<TH,TW<real,bool>,real,bool> = ⩛ HyperParameter = M.HyperParameter ⩛ Prior = M.Prior Gen = <$> fun (w,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 \( M_t \), as the Lebesgue-measurable subsets of \( V_t \), 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 \( f^{-1}(A) \in M_t \) for all \( A \in M_u \). Let \( \mathbb{M} \) be the set of finite measures on \( t \), that is, additive functions from \( M_t \) to the non-negative real numbers. Let the subprobability distributions \( \mathbb{S} \) be the finite measures whose range is \( [0,1) \). Let \( t \to u \) be the set of measure transformers from \( t \) to \( u \), defined as the partial functions \( M \to \mathbb{M} \). If \( \Gamma = \langle t_1, \ldots, t_m : t \rangle \) we let range(\( \Gamma \)) \( = t_1 \cdots t_m \). We make use of the following constructions on measures.

- The Dirac \( \delta \) measure is \( \delta_v(A) \triangleq 1 \) if \( v \in A \), 0 otherwise.
- Given a function \( f : t \to u \) and a measure \( \mu \in \mathbb{M} \), there is a measure \( \mu f^{-1} \in \mathbb{M} u \) 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 \( \mu \) on \( t \), a measurable set \( A \in M_t \) and a function \( f : t \to \mathbb{R} \), we write \( \int_A f d\mu \) or equivalently \( \int f(x) d\mu(x) \) for standard (Lebesgue) integration. This integration is always well-defined if \( \mu \) is finite and \( f \) is bounded.

- Given a measure \( \mu \) on \( t \), a function \( D \mu : t \to \mathbb{R} \) is a density for \( \mu \) iff \( (\mu(x) = \int_1(D \mu)(x) d\lambda(x) \) for all \( A \), where \( \lambda \) is the completion of the standard measure on \( t \) (which is built by taking products and disjoint sums of the counting measure on \( \mathbb{N} \) and the interval (Borel) measure on \( \mathbb{R} \)).

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:

- If \( \mu \in \mathbb{M} \), observe \( (t \to \mathbb{R}) \) (or \( t \to \mathbb{R} \)) extends the domain of a measure \( \mu \) using a function yielding sub-probability distributions. We let \( \operatorname{extend} \colon \mu \to \mathbb{M} \to \mathbb{M} \).
- The combinator \( \text{extend} \) composes the conditional density of a measure \( \mu \) over \( t \) on the event that an indicator function \( p \) of type \( t \to \mathbb{R} \) 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 \( B_r \) be the closed ball of radius \( r \) around \( 0 \).

We write \( \mu AB \triangleq \int \mathbb{B}_r m(x) \{ |x,y| \in AB \} \text{d}x \).

The combinator \( \text{observe} \) computes the conditional density of a measure \( \mu \) on \( t \) to \( \mathbb{R} \) when \( f : t \to \mathbb{R} \) is a measurable function yielding sub-probability distributions. We let \( \operatorname{observe} \colon \mu \to \mathbb{M} \to \mathbb{M} \).

- We define the Conditional Density: \( \mu[A|p = r] \triangleq \lim_{e \to 0} \mu(A \cap p^{-1}(B_r)) / \mu_p 1 \text{d}\lambda \) if the limit exists.

We observe \( p \mu A \triangleq \hat{\mu}[A|p = 0] \). As an example, if \( t = u \to \mathbb{R} \), \( p = \lambda(x,y), (y - c) \) and \( \mu \) has continuous density \( D \mu t \) then

\[
\operatorname{observe} \mu A = \int_{\{x \in |x,c|} D(\mu(x,d\lambda(x)) (5)
\]

and \( \int \hat{\mu}[A|p = x] d\lambda(x) = \mu(A) \). Notice that \( \operatorname{observe} p \mu A \) 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 \( (T_1 \to T_2) \mu A \triangleq (T_1(X \to \mu(X \uplus \mathbb{R}))) + (T_2(X \to \mu(X \uplus \mathbb{R}))) \). We also let \( \text{fail} \mu A \triangleq 0 \).

**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 3.6) we here extend observations to composite types. We write \( \text{obs} f \circ u \) if \( f : t \to u \), then the measure transformer \( \text{obs} f \) has type \( t \to u \). Below we let \( \mathbb{I}_1 = 1, \mathbb{I}_u = 1, \mathbb{I}_{u1} = 1, \mathbb{I}_{1u} = 1 \). We let \( \text{split} p = \text{true} \) if \( p(s) = s \) then \( \text{else} \) \( \text{inr} s \).

**Observations on Composite Types**: \( \text{obs} f \circ u \)

\[
\begin{align*}
\text{obs} f \circ \text{real} & \triangleq \text{observe} f \\
\text{obs} f \circ \text{int} & \triangleq \text{split} (\lambda.s.f(s) = 0) \quad \text{(pure id)} \quad \text{fail} \\
\text{obs} f \circ \mathbb{I} & \triangleq \text{pure id} \\
\text{obs} f \circ (1_u * u_2) & \triangleq \text{obs} (\text{and} \circ f) @ u_2 \quad \text{obs} (\text{or} \circ f) @ u_1 \\
\text{obs} f \circ (u_1 + u_2) & \triangleq \text{split} f \\
\text{obs} (\text{either id} \lambda_{| u_1}) & \circ f @ u_1 \\
\text{obs} (\text{either id} \lambda_{| u_2}) & \circ f @ u_2
\end{align*}
\]
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 \( \lambda_\bot \circ \text{int} = \text{fail} \).

**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 variable to the corresponding variables we use \( \text{pat} = O \) and \( \text{pat}(xx,x) := (\text{pat}(xx),x) \) as patterns and in return values. Below we define the measure transformer semantics of a program as \( \mathcal{N}(M) \) where \( xx \) are the free variables in the program. We use an auxiliary definition \( \mathcal{M}(M) \) for the semantics of the program \( M \) in the scope of \( y \). For closed terms \( M \) we obtain \( \mathcal{N}(M) \) by transforming the trivial probability measure.

Each signature \( \text{val} f : t_1 \times \cdots \times t_n 
\rightarrow t_{n+1} \) means that \( f \) is a total function with \( f : V_{t_1} \times \cdots \times V_{t_n} \rightarrow V_{t_{n+1}} \). If \( \text{Dist}(t) \rightarrow \text{Dist}(u) \) and \( V \in V_t \), there is a corresponding measure \( \text{Dist}(V) \). This measure is a probability measure for legal values of \( V \), and otherwise, since in 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 lift1: \( t \rightarrow (u_1 \vdash u_2) \rightarrow (t \vdash u_1) + (t \vdash u_2) \) defined as \( \lambda x,y.\text{match } y \text{ with } \text{inl } z : \text{inl}(z) \mid \text{inr } z : \text{inr}(z) \).

**Proposition 1** (Static Adequacy). If \( \Gamma \vdash M : t \) then \( \Gamma \vdash \text{dom}(\Gamma) \rightarrow (\text{range}(\Gamma) \times t) \).

As seen above, (observe \( p \)) \( M \) 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 \( \mu \) is compatible with respect to \( T \).

**Compatibility:** \( \mu \models T \quad (T \in t \rightarrow u \text{ and } \mu \in M(t)) \)

| \( \mu \models \text{pure } f \) | \( \mu \models \text{extend } m \) | \( \mu \models \text{fail} \) |
| \( \mu_1 \circ \mu_2 \models \text{lift1 } T1 \rightarrow T2 \) if \( \mu_1 \models T1 \) and \( \mu_2 \models T2 \) |
| \( \mu \models \text{lift1 } U \rightarrow U \) if \( \mu \models T \) and \( (T \models U) \) |

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

Compatibility is not necessary for \( T \mu \) to be defined.

**Hybrid Measure:**

\[
\begin{align*}
\text{let hybrid1} &= \text{if random(Bernoulli(0.5)) then 1.0 else random(Gaussian(0.0,1.0))} \\
\text{let hybrid2} &= \text{(random(Gaussian(0.0,0.9)),hybrid1)}
\end{align*}
\]

The measure \( \mu = \mathcal{N}(\text{hybrid2}) \) is the average of a two-dimensional Gaussian distribution and a line mass at \( y = 1.0 \). Here \( \mu \models \text{observe } \lambda(x,y),y \) but we do not have \( \mu \models \text{observe } \lambda(x,y),x \) 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 \( \text{observe } \lambda(x,y),x ) \cdot \mathcal{N}(\text{hybrid2}) = \gamma \cdot \phi(0.0) \cdot \mathcal{N}(0.0,\text{hybrid1}) \) where \( \phi(0.0) = 0.3989 \) 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 Pfenning 2002) which has semantics using the probability monad (Girard 1982). To treat fail we work in the sub-probability monad (Panangaden 1999), where the set of admissible distributions \( \mu \) also admits \( |\mu| < 1 \) (cf. the semantics of Barthe et al. (2012)).

The valuation \( \sigma \) maps the free variables of \( M \) to closed values.

**Monadic Semantics of Fun with fail:** \( \mathcal{P}(\mathcal{M}) \sigma \)

\[
\begin{align*}
|f(x)| &\models f(x)(A) \mu(x) &\text{Monadic bind} \\
|\text{return } v| &\models A \uparrow 1 \text{ if } v \in A, \text{ else } 0 &\text{Monadic return} \\
|\text{zero } A| &\models A \uparrow 0 &\text{Monadic zero} \\
|\mathcal{V}(\mathcal{P})| &\models \mathcal{V}(v) &\text{Monadic return} \\
|\text{Ilerite } (V_1,\ldots,V_n)| &\models \mathcal{V}(v_1) &\text{Ilerite } (V_1,\ldots,V_n) \\
|\text{match } V \text{ with } \text{inl } x : M |\models \text{inr } y : N \sigma &\models \mathcal{V}(v) &\text{either} \lambda v.\mathcal{P}(\mathcal{M}) \sigma (v,x \rightarrow v) (\lambda v.\mathcal{P}(\mathcal{M})(v,y \rightarrow v)) \\
|\text{random } (\mathcal{V}) \sigma | &\models \mathcal{P}(\mathcal{M}) \sigma &\models \mathcal{P}(\mathcal{M}) \sigma (v,x \rightarrow v) \\
|\text{fail }| &\models \text{zero} &
\end{align*}
\]

**Proposition 3.** If \( x_1 : t_1,\ldots,x_n : t_n \vdash M : t \) and \( \text{all observations in } M \text{ are fail, then} \)

\( \mathcal{M}(\text{all observations in } M \text { are fail) } \)

In particular, if \( \circ \vdash M : t \) we have \( \mathcal{M}(M) \o \).

**6.3 A Normal Form for Fun Expressions**

**Observation Translation:** \( \mathcal{O}(\mathcal{M}) \o (\text{does not appear in } N) \)

\[
\begin{align*}
|\text{observe } f(V)| &\models (f,V) \\
|\text{let } x = M \text{ in } N \sigma | &\models \text{let } x,o = \mathcal{O}(M) \text{ in } \mathcal{O}(N) \sigma (o',o) \\
|\text{match } V \text{ with } \text{inl } x : M_1 |\models \text{inr } y : M_2 \sigma &\models \text{match } V \text{ with } \text{inl } x \rightarrow \text{inr } y \rightarrow o \sigma \\
|\text{inl } x \rightarrow \text{inr } y \rightarrow o \sigma | &\models \text{inl } x \rightarrow \text{inr } y \rightarrow o \\
|\mathcal{M} | &\models \mathcal{M} \o &\text{otherwise}
\end{align*}
\]

**Proposition 4.** If \( \Gamma \vdash M : t \) then there is \( u \) such that \( \Gamma \vdash \mathcal{O}(M) : t \circ u \).

**Theorem 3.** If \( \Gamma \vdash M : t \) and \( N = \text{let } r.o = \mathcal{O}(\mathcal{M}) \text{ in } \text{observe } o.r \) and \( \mu \models \mathcal{N}(\text{dom}(\Gamma)) \text{ then } \mathcal{M}(\text{dom}(\Gamma)) \mu \models \mathcal{N}(\text{dom}(\Gamma)) \mu \).

**Corollary** (Normal form for closed terms). If \( \circ \vdash M : t \) and \( N \) is as above and \( \delta() \models \mathcal{N}(\mathcal{M}) \), then \( \mathcal{M}(M) = \mathcal{M}(N) \).
The compatibility \( M = \text{observe} (\text{first} \text{ (hybrid2)}) \) 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 \( \mathcal{H}[O/M] \) 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 \texttt{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 \texttt{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 \texttt{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:

\[
p(y_1^n|\{x_i\}_i=1^n, w, \overline{w}) = \prod_{i=1}^n \sum_{k=1}^K p(z = k|w) p(y_i|x_i, m_k, w_k) \tag{6}
\]

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

Mixtue Model Combinator

\[
\begin{align*}
\text{let} \; m &= \text{M(<HyperParameter = (uniform Beta, m1.HyperParameter, m2.HyperParameter >)}, \\
&\; \quad \text{Prior = <@ fun (bw,h1,h2) \to} \\
&\; \quad \quad (\text{random} (\text{Beta}(bw.trueCount,bw.falseCount)), \\
&\; \quad \quad (\%m1.Prior) h1, (\%m2.Prior) h2) @>) \\
&\; \quad \text{Gen=} <@ fun ((bias,w1,w2,x)) \to} \\
&\; \quad \quad \text{if breakSymmetry(random (Bernoulli(bias)))} \\
&\; \quad \quad \quad \text{then (}%m1.Gen( w1,x) \text{)} \\
&\; \quad \quad \quad \text{else (}%m2.Gen( w2,x) @>)
\end{align*}
\]

(The call to \texttt{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

\[
\begin{align*}
\text{let} \; m &= \text{M(GaussianModel,M,GaussianModel,M)}
\end{align*}
\]

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 \( \mathcal{E}(M) \triangleq \mathcal{A}[M] \), that is, the total measure of the semantics \( \mathcal{A}[M] \). Note that \( \mathcal{E}(M) \) may be different from \( 1 \) if \( M \) contains instances of \texttt{observe} or \texttt{fail}, as discussed in Section 6. More generally, if \( \Gamma \vdash M : t \), we let \( \mathcal{E}(M, \mu) \triangleq \mathcal{A}[M]_{\text{dom}(\Gamma)} \mu \). 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 \texttt{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.

\[
\mathcal{E}[M,N] \triangleq \begin{cases} \text{let } x = \text{random(Bernoulli(0.5)) in} \\
f \text{if } x \text{ then } M; (\text{else } N) ; () \\
\times \end{cases}
\]

Lemma 5. If \( \Gamma \vdash t \) and \( \Gamma \vdash N : u \) and \( \mu \in \mathcal{M} \text{ (range(\Gamma))} \) then

\[
\mathcal{E}(M, \mu) = \frac{\mathcal{A}[M,N]_{\text{dom}(\Gamma)} \mu (s, \text{true}) | s \in \text{Vrange(\Gamma)})}{\mathcal{E}(N, \mu) = \mathcal{A}[M,N]_{\text{dom}(\Gamma)} \mu (s, \text{false}) | s \in \text{Vrange(\Gamma)})}
\]

\textbf{Proof:} By expanding \( \mathcal{A}[M,N] \).

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 \( \phi \vdash M : t \) we have \( \mathcal{E}(M) = 2 \cdot \mathcal{A}[M,O] \) (true).

For open terms, if \( \Gamma \vdash M : t \) and \( \mu \in \mathcal{M} \text{ (range(\Gamma))} \) then \( \mathcal{E}(M, \mu) = 2 \cdot \mathcal{A}[M,O] \mu (s, \text{true}) | s \in \text{Vrange(\Gamma)}) \).

For this way of computing model evidence to work, it is critical that \texttt{observe} denotes \textit{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 \( \mathcal{A}[M,N] \) (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 \( \mathcal{A}[M,O] \) (true) < 0.5 for all \( M \). Frameworks based on Gibbs sampling such as BUGS also fail, since \texttt{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.

\[
p(y_1^n|\{x_i\}_i=1^n, w, \overline{w}) = \sum_{k=1}^K p(z = k|w) \prod_{i=1}^n p(y_i|x_i, m_k, w_k) \tag{7}
\]
We show below a combinator for the case $K = 2$. The parameter of the combined model includes a random boolean switch, which is inspected to generate each output $y_i$ from the model $m_{\text{switch}}$.

**Combinator for Model Averaging:**

```plaintext
let M (m1:Model<'TH1,'TW1,'TX,'TY>,
m2:Model<'TH2,'TW2,'TX,'TY>) :
  Model<('real='TH1 TH2),
    ('bool='TW1 TH2), '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 $m_1$ and $m_2$ (cf. Lemma 5), multiplied with the prior odds.

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

(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 $p(z|x, w)$ 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 $p(z|x_i, w)$ with prior $p(w|h)$ resulting in the combined sampling distribution

$$
p(y_i = y_i^* | x_i = x_i^*, w, \theta) = \prod_{i=1}^{n} \prod_{k=1}^{K} p(z = k | x_i, w) p(y_i | x_i, m_k, w_k)
$$

(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.

**Combinator for Mixture of Experts:**

```plaintext
let M (m:Model<('TH1,'TW1,'TX,'TY,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) h, (%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**

```plaintext
let m = ExpertMixture.M(Probit, 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 Filzbruch (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 $P_{RE}$, 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 $P_{RE}$ 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**

```
PRE(V, \sigma, M) \in R \cup \{-\infty, \bot\}
PRE(V, \sigma, U) = \log(1.0) if U = V else \log(0.0)
PRE(V, \sigma, f(V_1, \ldots, V_n)) = \log(1.0) if f(V_1, \ldots, V_n, \sigma) = V
PRE(V, \sigma, \bot) = \log(0.0)
PRE(V, \sigma, \text{match} U \text{ with } \text{inl} x : M | \text{inr} y : N) = PRE(V, \sigma, \text{match} U^\prime) if U = \text{inl} U^\prime
PRE(V, \sigma, \text{match} U \text{ with } \text{inr} y : M | \text{inr} y : N) = PRE(V, \sigma, \text{match} U^\prime) if U = \text{inr} U^\prime
PRE(V, \sigma, \text{let} x = M in N) = PRE(V, \sigma, x \mapsto U') if M is deterministic and M\sigma evaluates to U' \bot otherwise
PRE(V, \sigma, \text{fail}) = \log(0.0)
PRE(V, \sigma, observe f(V)) = \bot
```
When defined (that is, not ⊥), the \( \text{PRE}(V, \sigma, M) \) function computes the log-posterior density of the parameters (in \( \sigma \)) given the data \( V \).

**Theorem 5.** If \( \Gamma \vdash M : t \) and \( \sigma \) is a \( \Gamma \)-valuation and \( V \in V_t \) and \( \text{PRE}(V, \sigma, M) \neq \perp \) then \( \text{PRE}(V, \sigma, M) = \log ((D(\mathcal{P}[M] \sigma)(V))) \).

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 \( \text{PRE}(V_w, h \rightarrow V_w, \text{Prior}) \), where \( V_w \) is the hyperparameter, to the log-posterior density \( \text{PRE}(V_w, w \rightarrow V_w, M) \).

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

```haskell
let (Lambda(vh,p)) = m:Prior
let (Lambda(w,e)) =
  < Ω ⊗ fun w -> [[x for x in xs -> (%m.Gen)(w,x)]]] ⊗ >
let likelihood = "TW → real =
  (° fun w -> PRE(y,v,λ,w,f,e) + PRE(w,v,h,λ,p) *)
Compile(fun w -> Ω ⊗ (%P.RE(° y,v,Ω,η,v,w,e)) +
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 variant 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


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


