Distribution Transformer Semantics for Bayesian Machine Learning

Johannes Borgström Andrew D. Gordon Michael Greenberg
James Margetson Jurgen Van Gael
Microsoft Research

Abstract

The Bayesian approach to machine learning amounts to inferring posterior distributions of random variables from a probabilistic model of how the variables are related (that is, a prior distribution) and a set of observations of variables. There is a trend in machine learning towards expressing Bayesian models as probabilistic programs. As a foundation for this kind of programming, we propose a core functional calculus with primitives for sampling prior distributions and observing variables. We define a new set of combinators for distribution transformers, based on theorems in measure theory, and use these to obtain a rigorous semantics for our core calculus. Factor graphs are an important but low-level data structure in machine learning; they enable many efficient inference algorithms. We define a small imperative language that has a straight-forward semantics as factor graphs, which we then evaluate in an existing inference engine.

1. Introduction

In the past 15 years, statistical machine learning has unified many seemingly unrelated methods through the Bayesian paradigm. With a solid understanding of the theoretical foundations, advances in algorithms for inference, and numerous applications, the Bayesian paradigm is now the state of the art for learning from data. Still, one major limitation for communication and re-use of probabilistic models is a common language in which to describe them.

Csoft (Winn and Minka 2009) is an imperative language with an informal probabilistic semantics. Csoft is the native language of Infer.NET (Minka et al. 2009), a software library for Bayesian reasoning. A compiler turns Csoft programs into factor graphs (Kschischang et al. 2001), data structures that support efficient inference algorithms (Koller and Friedman 2009).

This paper borrows ideas from Csoft and extends them, placing the semantics on a firm footing.

Bayesian Models as Probabilistic Expressions Consider a simplified form of TrueSkill (Herbrich et al. 2007), a large-scale online system for ranking computer gamers. There is a population of players, each assumed to have a skill, which is a real number that cannot be directly observed. We observe skills only indirectly via a series of matches. The problem is to infer the skills of players given the outcomes of the matches.

Here is a concrete example: *Alice, Bob, and Cyd are new players. In a tournament of three games, Alice beats Bob, Bob beats Cyd, and Alice beats Cyd. What are their relative skills?*

In a Bayesian approach, we represent our uncertain knowledge of the skills as continuous probability distributions. The following probabilistic expression models our example by generating probability distributions for the players' skills.

```
// prior distributions, the hypothesis
let skill() = sample (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = sample (Gaussian(player,1.0))
observe (performance Alice > performance Bob) //Alice beats Bob
observe (performance Bob > performance Cyd) //Bob beats Cyd
observe (performance Alice > performance Cyd) //Alice beats Cyd
// return the skills
Alice,Bob,Cyd
```

A run of this expression would go as follows. We sample the skills of the three players from the *prior distribution* $\mathbf{Gaussian}(10.0, 20.0)$. Such a distribution can be pictured as a bell curve centred on 10.0, and gradually tailing off at a rate given by the *variance*, here 20.0. Sampling from such a distribution is a randomized operation that returns a real number, most likely close to the mean. For each match, the run continues by sampling an individual performance for each of the two players. Each performance is centered on the skill of a player, with low variance, making the performance closely correlated with but not identical to the skill. We then observe that the winner's performance is greater than the loser's. An *observation* **observe** M always returns $()$, but represents a constraint that M must hold. A whole run is valid if all encountered observations are true. The run terminates by returning the three skills.

Using Monte Carlo sampling, the expression generates a probability distribution for each of the skills based on running the expression many times, but keeping just the valid runs—the ones where the sampled skills correspond to the observed outcomes. We then compute the means of the resulting skills by applying standard statistical formulas. In the example above, we find that the *posterior distribution* of the returned skills has moved so that the mean of Alice's skill is greater than Bob's, which is greater than Cyd's.

Koller et al. (1997) pioneered the idea of writing Bayesian models as probabilistic programs and the idea has been gaining in popularity (Bonawitz 2008; Goodman et al. 2008; Kiselyov and Shan 2009; Park et al. 2005). To the best of our knowledge, all prior approaches apart from Csoft are based on precise inference using some form of Monte Carlo sampling. The use of factor graphs in Csoft additionally supports deterministic approximative inference algorithms, which are known to be significantly more efficient than sampling methods, where applicable.

In the example above, a drawn game would be modelled as the performance of two players being observed to be equal. Since these are randomly drawn from a continuous distribution, the probability of them actually being equal is zero, so we would not expect to see *any* valid runs. Observations with zero probability arise commonly in Bayesian models, and are supported by factor graphs, so our semantics (unlike any prior semantics) allows them.

Main Ideas of the Paper We propose Fun:

- Fun is a functional language for Bayesian models with primitives for probabilistic sampling and observing data.
- Fun supports collection types and collection comprehensions in order to express Bayesian models over large datasets.
- Fun has a rigorous probabilistic semantics based on measure-theoretic distribution transformers.
- Fun has an efficient implementation: our system compiles Fun to Imp, a subset of Csoft, and then relies on Infer.NET.

We designed Fun to be a subset of the F# dialect of ML (Syme et al. 2007a), for implementation convenience: F# reflection allows easy access to the abstract syntax of a program. Our support for collections is inspired by PQL (Van Gael et al. 2010), an experimental extension of SQL with features for defining factor graphs in terms of relational data. Our main technical contribution is a distribution transformer semantics, which supports models with discrete distributions, continuous distributions, and mixtures of the two.

As a substantial application, we supply distribution transformer semantics for Fun, Imp, and factor graphs, and use the semantics to verify the translations in our compiler. Our semantics is the first for languages, such as Fun or Imp, implemented by deterministic inference via factor graphs.

Plan of the Paper Section 2 introduces the idea of expressing a probabilistic model as code in a functional language, Fun, with primitives for generating and observing random variables. For a subset, Bernoulli Fun, limited to weighted Boolean choices, we describe in elementary terms an operational semantics that allows us to compute the conditional probability that the expression yields a given value given that the run was valid.

Section 3 describes a measure-theoretic denotational semantics for Fun as distribution transformers. Theorem 1 establishes agreement with the discrete semantics of Section 2 for Bernoulli Fun.

In Section 4, we turn to the efficient computation of the Fun semantics using factor graphs. Our strategy is to first compile Fun to a lower-level imperative language Imp (a subset of Csoft), and then to compile Imp to factor graphs. We introduce syntax for Imp and for factor graphs, and define the semantics of both using the distribution transformer semantics of Section 3. Theorem 2 and Theorem 3 establish the correctness of the first step, from Fun to Imp, and the second step, from Imp to factor graphs.

Section 5 extends our framework with collections. We extend Fun and Imp with arrays and array comprehensions, and extend the semantics, translations, and theorems of Section 4.

We describe our practical experience in Section 6. Our compiler directly implements the translation of Fun (including collections) to Imp by mapping F# models to Csoft, and then relies on Infer.NET for inference using factor graphs. All the examples in the paper have been executed with our system. As a case study, we extend our system with a front end to extract Fun models from a collection of PQL scripts, obtaining performance comparable to the custom implementation of PQL. This is evidence that large scale machine learning problems can be clearly expressed and efficiently implemented in Fun.

Section 7 describes related work, and Section 8 concludes.

A draft technical report, with additional listings and details of proofs, is available at `http://johannes.borgstroem.org/drafts/bayesTR.pdf`.

2. Bayesian Models as Probabilistic Expressions

We present a core calculus, Fun, for Bayesian reasoning via probabilistic functional programming with observations.

2.1 Syntax, Informal Semantics, and Bayesian Reading

Expressions are strongly typed, with types t built up from base scalar types b and pair types. We let c range over constant data of scalar type, n over integers and r over real numbers. We write $\mathrm{ty}(c) = t$ to mean that constant c has type t. For each base type b, we define a *zero element* 0_b.

Types, Constant Data, and Zero Elements:

$b ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real}$	Base types
$t ::= \mathbf{unit} \mid b \mid (t_1 * t_2)$	Compound types
$\mathrm{ty}(()) = \mathbf{unit} \quad \mathrm{ty}(\mathbf{true}) = \mathrm{ty}(\mathbf{false}) = \mathbf{bool}$	
$\mathrm{ty}(n) = \mathbf{int} \quad \mathrm{ty}(r) = \mathbf{real}$	
$0_{\mathbf{bool}} = \mathbf{true} \quad 0_{\mathbf{int}} = 0 \quad 0_{\mathbf{real}} = 0.0$	

We have arithmetic and Boolean operations on base types.

Signatures of Arithmetic and Logical Operators: $\otimes : t_1, t_2 \to t_3$

$\&\&, ||, = : \mathbf{bool}, \mathbf{bool} \to \mathbf{bool}$
$>, = : \mathbf{int}, \mathbf{int} \to \mathbf{bool}$
$+, -, *, \%, / : \mathbf{int}, \mathbf{int} \to \mathbf{int}$
$> : \mathbf{real}, \mathbf{real} \to \mathbf{bool}$
$+, -, * : \mathbf{real}, \mathbf{real} \to \mathbf{real}$

We have several standard probability distributions as primitive: $D : t \to u$ takes parameters in t and yields a random value in u. The names x_i below only document the meaning of the parameters.

Signatures of Distributions: $D : (x_1 : t_1 * \cdots * x_n : t_n) \to t_{n+1}$

$\mathbf{Bernoulli} : (\mathsf{success} : \mathbf{real}) \to \mathbf{bool}$
$\mathbf{Poisson} : (\mathsf{rate} : \mathbf{real}) \to \mathbf{int}$
$\mathbf{DiscreteUniform} : (\mathsf{max} : \mathbf{int}) \to \mathbf{int}$
$\mathbf{Gaussian} : (\mathsf{mean} : \mathbf{real} * \mathsf{variance} : \mathbf{real}) \to \mathbf{real}$
$\mathbf{Gamma} : (\mathsf{shape} : \mathbf{real} * \mathsf{scale} : \mathbf{real}) \to \mathbf{real}$

The expressions and values of Fun are below. Expressions are in a limited syntax akin to A-normal form, with let-expressions for sequential composition.

Fun: Values and Expressions

$V ::= x \mid c \mid (V, V)$	Value
$M, N ::=$	Expression
$\quad V$	value
$\quad V_1 \otimes V_2$	arithmetic or logical operator
$\quad V.1$	left projection from pair
$\quad V.2$	right projection from pair
$\quad \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2$	conditional
$\quad \mathbf{let}\ x = M\ \mathbf{in}\ N$	let (scope of x is N)
$\quad \mathbf{sample}\ (D(V))$	sample a distribution
$\quad \mathbf{observe}\ V$	observation

We outline an intuitive sampling semantics for this language; the formal details and a more general semantics come later. Let a run of a closed expression M be the process of evaluating M to a value. The evaluation of most expressions is standard, apart from sampling and observation:

- To run $\mathbf{sample}\ (D(V))$, where $V = (c_1, \ldots, c_n)$, choose a value c according to distribution $D(c_1, \ldots, c_n)$, and return c.
- To run $\mathbf{observe}\ V$, always return $()$. We say the observation is *valid* if and only if the value V is some zero element 0_b.

Due to the presence of sampling, different runs of the same expression may yield more than one value, with differing probabilities. Let a run be *valid* so long as every encountered observation is valid. The overall semantics of an expression is the conditional probability of returning a particular value, given a valid run.

Example: Two Coins, Not Both Tails

```
let heads1 = sample (Bernoulli(0.5)) in
let heads2 = sample (Bernoulli(0.5)) in
let u = observe (heads1 || heads2) in
(heads1,heads2)
```

There are four distinct runs, each with probability $1/4$, corresponding to the possible combinations of Booleans heads1 and heads2. All these runs are valid, apart from the one for heads1 = **false** and heads2 = **false** (representing two tails), since the observation **observe**(**false**||**false**) is not valid. The overall semantics of this expression is a conditional probability distribution assigning probability $1/3$ to the values (**true**, **false**), (**false**, **true**), and (**true**, **true**), but probability 0 to the value (**false**, **false**).

Our semantics allows us to interpret an expression as a Bayesian model. We interpret the distribution of possible return values as the *prior probability* of the model. The constraints on valid runs induced by observations represent new evidence or training data. The conditional probability of a value given a valid run is the *posterior probability*: an adjustment of the prior probability given the evidence or training data.

In particular, the expression above can be read as a Bayesian model of the problem: *I toss two coins. I observe that not both are tails. What is the probability of each outcome?* The uniform distribution of two Booleans represents our prior knowledge about two coins, the **observe** expression represents the evidence that not both are tails, and the overall semantics is the posterior probability of two coins given this evidence.

In the remainder of this section, we define the type system for Fun, define a formal semantics for the discrete subset of Fun, and describe further examples. Our discrete semantics is a warm up before Section 3. There we deploy measure theory to give a semantics to our full language, that allows both discrete and continuous prior distributions.

2.2 Syntactic Conventions and Monomorphic Typing Rules

We recite our standard syntactic conventions and typing rules.

We identify phrases of syntax (such as expressions) up to consistent renaming of bound variables (such as x in a let-expression). Let $\mathrm{fv}(\phi)$ be the set of free variables occurring in phrase ϕ. Let $\phi\{\psi/x\}$ be the outcome of substituting phrase ψ for each free occurrence of variable x in phrase ϕ. To keep our core calculus small, we treat function definitions as macros with call-by-value semantics. In particular, in examples, we write first-order non-recursive function definitions in the form **let** $f\ x_1\ \ldots\ x_n = M$, and we allow function applications $f\ M_1\ \ldots\ M_n$ as expressions. We consider such a function application as being a shorthand for the expression **let** $x_1 = M_1$ **in** $\ldots$**let** $x_n = M_n$ **in** M, where the bound variables $x_1, \ldots, x_n$ do not occur free in $M_1, \ldots, M_n$. We allow expressions to be used in place of values, via insertion of suitable let-expressions. For example, (M_1, M_2) stands for **let** $x_1 = M_1$ **in let** $x_2 = M_2$ **in** (x_1, x_2), and $M_1 \otimes M_2$ stands for **let** $x_1 = M_1$ **in let** $x_2 = M_2$ **in** $x_1 \otimes x_2$, when either M_1 or M_2 or both is not a value. Let $M_1; M_2$ stand for **let** $x = M_1$ **in** M_2 where $x \notin \mathrm{fv}(M_2)$. The notation $t = t_1 * \cdots * t_n$ for tuple types means the following: when $n = 0$, $t =$ **unit**; when $n = 1$, $t = t_1$; when $n = 2$, t is the primitive pair type $t_1 * t_2$; and when $n > 2$, $t = t_1 * (t_2 * \cdots * t_n)$. In listings, we rely on syntactic abbreviations available in F#, such as layout conventions (to suppress **in** keywords) and writing tuples as $M_1, \ldots, M_n$ without enclosing parentheses.

Let a *typing environment*, Γ, be a list of the form $\varepsilon, x_1 : t_1, \ldots, x_n : t_n$; we say Γ is *well-formed* and write $\Gamma \vdash \diamond$ to mean that the variables x_i are pairwise distinct.

Typing Rules for Fun Expressions: $\Gamma \vdash M : t$

$$\text{(Fun Var)}\ \frac{\Gamma \vdash \diamond \quad (x:t) \in \Gamma}{\Gamma \vdash x : t} \qquad \text{(Fun Const)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathrm{ty}(c)} \qquad \text{(Fun Pair)}\ \frac{\Gamma \vdash V_1 : t_1 \quad \Gamma \vdash V_2 : t_2}{\Gamma \vdash (V_1, V_2) : t_1 * t_2}$$

$$\text{(Fun Operator)}\ \frac{\otimes : t_1, t_2 \to t_3 \quad \Gamma \vdash V_1 : t_1 \quad \Gamma \vdash V_2 : t_2}{\Gamma \vdash V_1 \otimes V_2 : t_3} \qquad \text{(Fun Proj1)}\ \frac{\Gamma \vdash V : t_1 * t_2}{\Gamma \vdash V.1 : t_1} \qquad \text{(Fun Proj2)}\ \frac{\Gamma \vdash V : t_1 * t_2}{\Gamma \vdash V.2 : t_2}$$

$$\text{(Fun If)}\ \frac{\Gamma \vdash V : \textbf{bool} \quad \Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t}{\Gamma \vdash \textbf{if } V \textbf{ then } M_1 \textbf{ else } M_2 : t} \qquad \text{(Fun Let)}\ \frac{\Gamma \vdash M_1 : t_1 \quad \Gamma, x : t_1 \vdash M_2 : t_2}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2}$$

$$\text{(Fun Sample)}\ \frac{D : (x_1 : t_1 * \cdots * x_n : t_n) \to t_{n+1} \quad \Gamma \vdash V : (t_1 * \cdots * t_n)}{\Gamma \vdash \textbf{sample } (D(V)) : t_{n+1}} \qquad \text{(Fun Observe)}\ \frac{\Gamma \vdash V : b}{\Gamma \vdash \textbf{observe } V : \textbf{unit}}$$

2.3 Formal Semantics for Bernoulli Fun

Let Bernoulli Fun be the fragment of our calculus where every **sample** expression takes the form **sample** (**Bernoulli**(c)) for some real $c \in (0,1)$, that is, a weighted Boolean choice returning **true** with probability c, and **false** with probability $1-c$. We show that closed well-typed expressions induce conditional probabilities $P(\mathsf{value} = V \mid \mathsf{valid})$, the probability that the value of a valid run of E is V.

For this calculus, we inductively define an operational semantics, $M \to^p M'$, meaning that expression M takes a step to M' with probability p.

Reduction Relation: $M \to^p M'$ **where** $p \in (0,1]$

$V_1 \otimes V_2 \to^1 c$ if $V_1 = c_1$, $V_2 = c_2$, and $c = c_1 \otimes c_2$
$(V_1, V_2).1 \to^1 V_1$
$(V_1, V_2).2 \to^1 V_2$
if true then M_1 **else** $M_2 \to^1 M_1$
if false then M_1 **else** $M_2 \to^1 M_2$
let $x = V$ **in** $M \to^1 M\{V/x\}$

$\mathcal{R}[M] \to^p \mathcal{R}[M']$ if $M \to^p M'$ for reduction context $\mathcal{R}$ given by $\mathcal{R} ::= [\,] \mid$ **let** $x = \mathcal{R}$ **in** M

sample (**Bernoulli**(c)) $\to^c$ **true** if $c \in (0,1)$
sample (**Bernoulli**(c)) $\to^{1-c}$ **false** if $c \in (0,1)$
observe $V \to^1 ()$

Since there are no features for recursion or unbounded iteration, there are no non-terminating reduction sequences $M_1 \to^{p_1} \ldots M_n \to^{p_n} \ldots$. Moreover, we can prove standard preservation and progress lemmas.

We consider a fixed expression M such that $\varepsilon \vdash M : t$.

Let the space Ω be the set of all runs of M, where a *run* is a sequence $\omega = (M_1, \ldots, M_{n+1})$ for $n \geq 0$ and p_1, $\ldots$, p_n such that $M = M_1 \to^{p_1} \cdots \to^{p_n} M_{n+1} = V$; we define the functions $\mathsf{value}(\omega) = V$ and $\mathsf{prob}(\omega) = 1 p_1 \ldots p_n$, and we define the predicate $\mathsf{valid}(\omega)$ to hold if and only if whenever $M_j = \mathcal{R}[\textbf{observe } V]$ then $V = 0_b$ for some zero element 0_b. Since M is well-typed, is normalizing, and samples only from Bernoulli distributions, Ω is finite.

Let an *event*, α or β, be a subset of Ω. Let α and β range over events, and let probability $P(\alpha) = \sum_{\omega \in \alpha} \mathsf{prob}(\omega)$.

PROPOSITION 1 *The function $P(\alpha)$ forms a* probability distribution*, that is, (1) we have $P(\alpha) \geq 0$ for all α, (2) $P(\Omega) = 1$, and (3) $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$ if $\alpha \cap \beta = \varnothing$.*

To give the semantics of our expression M we first define the following probabilities and events. Given a value V, $\mathsf{value} = V$ is the event $\mathsf{value}^{-1}(V) = \{\omega \mid \mathsf{value}(\omega) = V\}$. Hence, $P(\mathsf{value} = V)$ is the odds (or *prior probability*) that a run terminates with V. We let the event $\mathsf{valid} = \{\omega \in \Omega \mid \mathsf{valid}(\omega)\}$; hence, $P(\mathsf{valid})$ is the probability that a run is valid.

If $P(\beta) \neq 0$, the *conditional probability of α given β* is

$$P(\alpha \mid \beta) \triangleq \frac{P(\alpha \cap \beta)}{P(\beta)}$$

The semantics of a program is given by the conditional probability distribution $P(\mathsf{value} = V \mid \mathsf{valid}) = P((\mathsf{value}^{-1}(V)) \cap \mathsf{valid})/P(\mathsf{valid})$, the conditional probability that a run returns V given a valid run, also known as the *posterior probability*.

The conditional probability $P(\mathsf{value} = V \mid \mathsf{valid})$ is only defined when $P(\mathsf{valid})$ is not zero. For pathological choices of M such as **observe false** or **let** $x = 3$ **in observe** $x = 2$ there are no valid runs, so $P(\mathsf{valid}) = 0$, and $P(\mathsf{value} = V \mid \mathsf{valid})$ is undefined. (This is an occasional problem in practice; Bayesian inference engines such as Infer.NET fail in this situation with a zero-probability exception.)

2.4 Example in Bernoulli Fun

The expression below encodes the question: *1% of a population have a disease. 80% of subjects with the disease test positive, and 9.6% without the disease also test positive. If a subject is positive, what are the odds they have the disease?* (Yudkowsky 2003)

Epidemiology: Odds of Disease Given Positive Test

```
let has_disease = sample (Bernoulli(0.01))
let positive_result = if has_disease
                      then sample (Bernoulli(0.8))
                      else sample (Bernoulli(0.096))
observe positive_result
has_disease
```

For this expression, we have $\Omega = \{\omega_{tt}, \omega_{tf}, \omega_{ft}, \omega_{ff}\}$ where each run $\omega_{c_1 c_2}$ corresponds to the choice $\mathsf{has\_disease} = c_1$ and $\mathsf{positive\_result} = c_2$. The probability of each run is:

- $\mathsf{prob}(\omega_{tt}) = 0.01 \times 0.8 = 0.008$ (true positive)
- $\mathsf{prob}(\omega_{tf}) = 0.01 \times 0.2 = 0.002$ (false negative)
- $\mathsf{prob}(\omega_{ft}) = 0.99 \times 0.096 = 0.09504$ (false positive)
- $\mathsf{prob}(\omega_{ff}) = 0.99 \times 0.904 = 0.89496$ (true negative)

The semantics $P(\mathsf{value} = \textbf{true} \mid \mathsf{valid})$ here is the conditional probability of having the disease, given that the test is positive.

Here $P(\mathsf{valid}) = \mathsf{prob}(\omega_{ft}) + \mathsf{prob}(\omega_{tt})$ and $P(\mathsf{value} = \textbf{true} \cap \mathsf{valid}) = \mathsf{prob}(\omega_{tt})$, so $P(\mathsf{value} = \textbf{true} \mid \mathsf{valid}) = 0.008/(0.008 + 0.09504) = 0.07764$. So the likelihood of disease given a positive test is just 7.8%, less than one might think.

This example illustrates inference on an explicit enumeration of the runs in Ω. The size of Ω is exponential in the number of **sample** expressions, so although illustrative, this style of inference does not scale up. As we explain in Section 4, our implementation strategy is to translate Fun expression to factor graphs, for efficient approximate inference.

3. Semantics as Distribution Transformers

When we want to generalize the semantics shown in the previous section to continous distributions (such as the Gaussian distribution, also called "normal"), we run into the problem that the probability of any particular value being drawn from such a distribution is zero. Ramsey and Pfeffer (2002) address this problem by working with monads of probability distributions, where the meaning of a program is a distribution of its possible outcomes. We can generalize the semantics of **observe** as filtering the set of runs to this setting (yielding what we may call a sub-probability monad), as long as the events that are being observed have non-zero probability. However, in machine learning, events that have zero probability are regularly observed.

A common, simple probabilistic model is the naive Bayesian classifier. In the training phase, it is given objects together with their classes and the values of their pertinent features. Below, we show the training for a single feature: the weight of the object. The zero probability events are weight measurements, that are assumed to follow a normal distribution. The outcome of the training is the posterior weight distributions for the different classes.

```
let wPrior() = sample (Gaussian(0.5,1.0))
let Glass,Watch,Plate = wPrior(),wPrior(),wPrior()
let weight objClass objWeight =
    observe (objWeight−(sample (Gaussian(objClass,1.0))))
weight Glass .18; weight Glass .21
weight Watch .11; weight Watch .073
weight Plate .23; weight Plate .45
Watch,Glass,Plate
```

The example uses **observe** (x−y) to denote that the difference between the weights x and y is 0. The reason for not instead writing x=y is that conditioning on events of zero probability without specifying the random variable they are drawn from is not in general well-defined (cf. Borel's paradox (Jaynes 2003)). To avoid this, we instead observe the random variable x−y, at the value 0. To give a formal semantics to such observations, as well as to mixtures of continous and discrete distributions, we turn to measure theory. Our treatment of measure theory and probability follows Billingsley (1995) and Rosenthal (2006).

3.1 Types as Measurable Spaces

In the remainder of the paper, we let Ω range over sets of possible outcomes; examples of Ω in our semantics include $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$, $\mathbb{Z}$, and $\mathbb{R}$. A *σ-algebra* over Ω is a set $\mathcal{M} \subseteq \mathcal{P}(\Omega)$ which (1) contains $\varnothing$ and Ω and (2) is closed under complement and countable union and intersection. A *measurable space* is a pair $(\Omega, \mathcal{M})$ where $\mathcal{M}$ is a σ-algebra over Ω. We use the notation $\sigma_\Omega(S)$, when $S \subseteq \mathcal{P}(\Omega)$, for the smallest σ-algebra over Ω that is a superset of S; we may omit Ω when it is clear from context. If $(\Omega, \mathcal{M})$ and $(\Omega', \mathcal{M}')$ are measurable spaces, then $f : \Omega \to \Omega'$ is measurable if and only if for all $A \in \mathcal{M}'$, $f^{-1}(A) \in \mathcal{M}$.

We give each first-order type t an interpretation as a measurable space $\mathcal{T}[\![t]\!] \triangleq (V_t, \mathcal{M}_t)$ below. We write () for $\varnothing$, the unit value.

Semantics of Types as Measurable Spaces:

$$\mathcal{T}[\![\textbf{unit}]\!] = (\{()\}, \{\{()\}, \varnothing\})$$
$$\mathcal{T}[\![\textbf{bool}]\!] = (\mathbb{B}, \mathcal{P}(\mathbb{B}))$$
$$\mathcal{T}[\![\textbf{int}]\!] = (\mathbb{Z}, \mathcal{P}(\mathbb{Z}))$$
$$\mathcal{T}[\![\textbf{real}]\!] = (\mathbb{R}, \sigma_{\mathbb{R}}(\{[a,b] \mid a,b \in \mathbb{R}\}))$$
$$\mathcal{T}[\![t * u]\!] = (V_t \times V_u, \sigma_{V_t \times V_u}(\{m \times n \mid m \in \mathcal{M}_t,\ n \in \mathcal{M}_u\}))$$

The set $\sigma_{\mathbb{R}}(\{[a,b] \mid a,b \in \mathbb{R}\})$ in the definition of $\mathcal{T}[\![\textbf{real}]\!]$ is the Borel σ-algebra on the real line, which is the smallest σ-algebra containing all closed (and open) intervals. Below, we write $f : t \to u$ to denote that $f : V_t \to V_u$ is measurable, that is, that $f^{-1}(B) \in \mathcal{M}_t$ for all $B \in \mathcal{M}_u$.

3.2 Distributions as Finite Measures

A *finite measure* μ on a measurable space $(\Omega, \mathcal{M})$ is a function $\mathcal{M} \to \mathbb{R}^+$ that is countably additive, that is, if the sets $A_0, A_1, \ldots \in \mathcal{M}$ are pairwise disjoint, then $\mu(\cup_i A_i) = \sum_i \mu(A_i)$. All the measures we consider in this paper are finite. Let $\mathtt{D}\,t$, the set of *distributions on type* t, be the set of finite measures on $\mathcal{T}[\![t]\!]$; these correspond closely to computations in the probability monad. We make use of the following constructions on measures.

- Given a measurable function $f : t \to u$ and a measure $\mu \in \mathtt{D}\,t$, we get a measure $\mu f^{-1} \in \mathtt{D}\,u$ given by $(\mu f^{-1})(B) \triangleq \mu(f^{-1}(B))$.
- Given a finite measure μ and a measurable set B, we let $\mu|_B(A) \triangleq \mu(A \cap B)$ be the restriction of μ to B.
- We can add two measures on the same set as $(\mu_1 + \mu_2)(A) \triangleq \mu_1(A) + \mu_2(A)$.

Some Standard Distributions In the discrete case it is sufficient to define the probabilities of singleton sets; all others follow from countable additivity.

Discrete Probability Distributions:

Bernoulli(p) **true** $\triangleq p$ and **Bernoulli**(p) **false** $\triangleq 1-p$
Poisson(l) $n \triangleq e^{-l} l^n / n!$
DiscreteUniform(m) $i \triangleq 1/m$ if $0 \le i < n$, 0 otherwise.

The named continous distributions have a *probability density* with respect to standard (Lebesgue) integration on $\mathbb{R}$. The function $\dot{\mu}$ is a density for μ iff $\mu(A) = \int_A \dot{\mu}\, dx$ for all measurable A.

Densities of Continuous Distributions:

Gaussian(m, v) $x \triangleq e^{-(x-m^2)/2v^2} / \sqrt{2\pi v^2}$
Gamma(s, p) $x \triangleq x^{s-1} e^{-px} p^s / \Gamma(s)$

The Dirac δ distribution is defined for all base types and is given by $\delta_c(A) \triangleq 1$ if $c \in A$, 0 otherwise. We write δ for $\delta_{0.0}$.

3.3 Distribution Transformers

A *distribution transformer* is a function from distributions to distributions, that is, from finite measures to finite measures. We let $t \rightsquigarrow u$ be the set of functions $\mathtt{D}\,t \to \mathtt{D}\,u$. We rely on the following combinators on distribution transformers in our formal semantics.

Distribution Transformer Combinators:

$\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$
$\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$
$\mathtt{choose} \in (V_t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$
$\mathtt{extend} \in (V_t \to \mathtt{D}\,u) \to (t \rightsquigarrow (t * u))$
$\mathtt{observe} \in (t \to u) \to (t \rightsquigarrow t)$

To lift a pure function to a distribution transformer, we use $\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$: given $f : t \to u$, we let $\mathtt{pure}\ f\ \mu\ A \triangleq \mu f^{-1}(A)$.

To sequentially compose two distribution transformers, we define $\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$ as $T \ggg U \triangleq U \circ T$.

The combinator $\mathtt{choose} \in (t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$ makes a conditional choice between distribution transformers, if its first argument has finite range.

We let $\mathtt{choose}\ K \mu\ B \triangleq \sum_{T \in \mathrm{range}(K)} T(\mu|_{K^{-1}(T)})(B)$.

The combinator $\mathtt{extend} \in (V_t \to \mathtt{D}\,u) \to (t \rightsquigarrow (t * u))$ extends the domain of a measure using a function yielding distributions.

We let $\mathtt{extend}\ m\ \mu\ AB \triangleq \int_{V_t} m(x)(\{y \mid (x,y) \in AB\}) d\mu(x)$.

Observation as a Distribution Transformer The combinator $\mathtt{observe} \in (t \to b) \to (t \rightsquigarrow t)$ restricts a measure over $\mathcal{T}[\![t]\!]$ to the event that an indicator function is zero. We here consider restriction as *unnormalized* conditioning of a measure on an event. In general, this is defined as follows.

Given a finite measure μ on $\mathcal{T}[\![t]\!]$ and a function $p : t \to b$, we consider the family of events $p(x) = c$. There exists a family of finite measures $\mu[\cdot || p = c]$ on $\mathcal{T}[\![b]\!]$ with the property that for all $B \in \mathcal{M}_u$,

$$\int_B \mu[A||p = x]\, d(\mu p^{-1})(x) = \mu(A \cap p^{-1}(B)).$$

Given such a family, we let $\mathtt{observe}\ p\ \mu\ A \triangleq \mu[A||p = 0_b]$.

Versions of conditional probability. We now fix c, letting $C = p^{-1}(c)$, and consider the measure $\nu(A) = \mu[A||p = c]$. It is uniquely defined (with $\nu(A) = \mu|_C(A)$) if $\mu(C) \neq 0$. If $\mu(C) = 0$ this is not the case; two versions of $\mu[\cdot||p = \cdot]$ may differ on a set B with $\mu p^{-1}(B) = 0$. However, we can give a unique definition in certain cases. Assume that $\mu(C) = 0$. Case (1): If t or u is discrete, we then require that $\mu[A||p = c] = 0$ for all A. Case (2): If μ has a continuous density $\dot{\mu}$ on some neighbourhood of C we require that

$$\mu[A||p = c] = \int_A \delta_c(p(\vec{x}))\dot{\mu}(\vec{x})\, d\vec{x}.$$

For the remainder of the paper, we assume that all $\mu[A||p = c]$ satisfy (1) and (2) above. If $T : t \rightsquigarrow u$ is a distribution transformer composed from the primitives above and $\mu \in \mathtt{D}\,t$, we say that T is compatible with μ if every application of $\mathtt{observe}\ f$ to some μ' in the evaluation of $T(\mu)$ satisfies the preconditions of (1) or (2) above (i.e., f is either discrete or μ has a continuous density on some neighbourhood of $f^{-1}(0.0)$).

As an example, if $p : t \to$ **bool** is a predicate on values of type t, we have $\mathtt{observe}\ p\ \mu\ A = \mu(A \cap \{x \mid p(x) =$ **true**$\})$. In the continuous case, if $V_t = \mathbb{R} \times \mathbb{R}^k$, $p = \lambda(y, \vec{x}).(y - c)$ and μ has density D_μ then

$$\mathtt{observe}\ p\ \mu\ A = \int_A \delta(y-c) D_\mu(y,\vec{x})\, d(y,\vec{x}) = \int_{\{\vec{x} \mid (c,\vec{x}) \in A\}} D_\mu(c,\vec{x})\, d\vec{x}$$

3.4 Distribution Transformer Semantics of Fun

In order to give a compositional denotational semantics of Fun programs, we give a semantics to open programs, later to be placed in some closing context. Since observations change the distributions of program variables, we can draw a parallel between open Fun terms and programs in ML with references. In the latter setting, we can give a denotation to a program as a function from valuations of reference cells to a return value and a reference valuation. Similarly, we give semantics to an open Fun term by mapping a distribution over assignments to the term's free variables to a joint distribution of the term's return value and assignments to its free variables. This choice was inspired by the semantics of pWHILE (Barthe et al. 2009). First, we define a data structure for an evaluation environment assigning values to variable names, and corresponding operations.

Given a type environment $\Gamma = x_1{:}t_1, \ldots, x_n{:}t_n$, we let $\mathtt{S}\langle\Gamma\rangle$ be the set of states, or finite maps $s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$ such that for all $i = 1, \ldots, n$, $\varepsilon \vdash V_i : t_i$. If $n \ge 1$ we let $\mathcal{T}[\![\mathtt{S}\langle\Gamma\rangle]\!] \triangleq \mathcal{T}[\![t_1 * \cdots * t_n]\!]$ be the measurable space of states in $\mathtt{S}\langle\Gamma\rangle$. We define $\mathrm{dom}(s) = \{x_1, \ldots, x_n\}$. We define the following operators on states.

Operations on States:

$\mathtt{add}\ x\ (s, V) \triangleq s \cup \{x \mapsto V\}$	if $\varepsilon \vdash V : t$ and $x \notin \mathrm{dom}(s)$, s otherwise.
$\mathtt{lookup}\ x\ \ s \triangleq s(x)$	if $x \in \mathrm{dom}(s)$, () otherwise.
$\mathtt{drop}\ X\ s \triangleq \{(x \mapsto V) \in s \mid x \notin X\}$	

We now use these combinators to give a semantics to Fun programs as distribution transformers. We assume that all bound variables in a program are different from the free variables and each other. Below, $\mathcal{V}[\![V]\!]\ s$ gives the valuation of V in state s.

Distribution Transformer Semantics of Fun:

$\mathcal{V}[\![x]\!]\ s \triangleq \texttt{lookup}\ x\ s$
$\mathcal{V}[\![c]\!]\ s \triangleq c$
$\mathcal{V}[\![(V_1, V_2)]\!]\ s \triangleq (\mathcal{V}[\![V_1]\!]\ s, \mathcal{V}[\![V_2]\!]\ s)$

$\mathcal{A}[\![V]\!] \triangleq \texttt{pure}\ \lambda s.(s, \mathcal{V}[\![V]\!]\ s)$
$\mathcal{A}[\![V_1 \otimes V_2]\!] \triangleq \texttt{pure}\ \lambda s.(s, ((\mathcal{V}[\![V_1]\!]\ s) \otimes (\mathcal{V}[\![V_2]\!]\ s)))$
$\mathcal{A}[\![V.1]\!] \triangleq \lambda s.(s, \texttt{fst}\ \mathcal{V}[\![V]\!]\ s)$
$\mathcal{A}[\![V.2]\!] \triangleq \lambda s.(s, \texttt{snd}\ \mathcal{V}[\![V]\!]\ s)$
$\mathcal{A}[\![\textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N]\!] \triangleq$
$\quad \texttt{choose}\ \lambda s.\texttt{if}\ \mathcal{V}[\![V]\!]\ s\ \texttt{then}\ \mathcal{A}[\![M]\!]\ \texttt{else}\ \mathcal{A}[\![N]\!]$

$\mathcal{A}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] \triangleq \mathcal{A}[\![M]\!] \ggg \texttt{pure}\ (\texttt{add}\ x) \ggg$
$\quad \mathcal{A}[\![N]\!] \ggg \texttt{pure}\ \lambda(s,y).((\texttt{drop}\ \{x\}\ s), y))$

$\mathcal{A}[\![\textbf{sample}(D(V))]\!] \triangleq \texttt{extend}\ \lambda s.D(\mathcal{V}[\![V]\!]\ s)$

$\mathcal{A}[\![\textbf{observe}\ V]\!] \triangleq (\texttt{observe}\ \lambda s.\mathcal{V}[\![V]\!]\ s) \ggg \texttt{pure}\ \lambda s.(s, ())$

A value expression V returns the valuation of V in the current state, which is left unchanged. Similarly, binary operations and projections have a deterministic meaning given the current state. An **if** V expression runs the distribution transformer given by the **then** branch on the states where V evaluates true, and the transformer given by the **else** branch on all other states, using the combinator `choose`. The expression **let** $x = M$ **in** N intuitively first runs M and binds its return value to x using `add`. After running N, the binding is discarded using `drop`. When sampling, **sample**$(D(V))$ extends the state distribution with a value drawn from the distrinbution D, with parameters V depending on the current state. An observation **observe** V modifies the current distribution by restricting it to states where V is zero. It is implemented with the `observe` combinator, and it always returns the unit value.

LEMMA 2 *If* $s : \mathtt{S}\langle\Gamma\rangle$ *and* $\Gamma \vdash V : t$ *then* $\mathcal{V}[\![V]\!]\ s \in V_t$.

LEMMA 3 *If* $\Gamma \vdash M : t$ *then* $\mathcal{A}[\![M]\!] \in \mathtt{S}\langle\Gamma\rangle \rightsquigarrow (\mathtt{S}\langle\Gamma\rangle * t)$.

The distribution transformer semantics of Fun are hard to use directly, except in the case of Bernoulli Fun where they can be directly implemented: a naive implementation of $\mathtt{D}\langle\mathtt{S}\langle\Gamma\rangle\rangle$ is as a map assigning a probability to each possible variable valuation. In this simple case, the distribution transformer semantics of closed programs also coincides with the sampling semantics.

THEOREM 1 *Suppose* $\varepsilon \vdash M : t$ *for some M in Bernoulli Fun. If* $\mu = \mathcal{A}[\![M]\!]\ \delta_{()}$ *and* $\varepsilon \vdash V : t$ *then* $P(\mathsf{value} = V \mid \mathsf{valid}) = \mu(\{V\})/\mu(V_t)$.

For this theorem to hold, it is critical that `observe` denotes unnormalized conditioning (filtering); otherwise the two program fragments **observe** (x=y) and **if** x **then observe** (y=**true**) **else observe** (y=**false**) that have the same operational semantics would have different distribution transformer semantics.

4. Semantics as Factor Graphs

A naive implementation of the distribution transformer semantics of the previous section would work directly with distributions of states, which could be exponential in the number of variables in scope. For large models, this becomes intractable. In this section, we instead give a semantics to Fun programs as *factor graphs* (Kschischang et al. 2001), which will be of linear size in the size of the program. We define this semantics in two steps: First, we compile the Fun program into a simple imperative language Imp, which has a straight-forward semantics as factor graphs. The implementation advantage of translating Fun to Imp, over simply generating factor graphs directly from Fun (cf. McCallum et al. (2009)), is that the translation preserves the structure of the input model, which can be exploited by inference algorithms. This is particularly useful when we add loops in Section 5. This allows us to choose from among a variety of factor graph inference engines to infer distributions from Imp programs (and via Theorem 2, the same applies to Fun).

4.1 Imp: An Imperative Core Calculus

Imp is an imperative language, based on the static single assignment intermediate form. It is a sublanguage of Csoft, the input language of Infer.NET (Minka et al. 2009), and is intended to have a simple semantics as a factor graph. A statement in the language either stores the result of a primitive operation in a location; observes a location to be zero, or branches on the value of a location. Imp shares the base types b with Fun, but has no tuples.

Syntax of Imp:

$L ::= () \mid l, l', \ldots$	Locations
$E, F ::=$	Expression
$\quad c$	constant
$\quad l$	location
$\quad l \otimes l$	binary operation
$I ::=$	Statement
$\quad l \leftarrow E$	assignment
$\quad l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)$	sampling
$\quad \textbf{observe}_b\ l$	observation
$\quad \textbf{if}\ l\ \textbf{then}_{S_1}\ C_1\ \textbf{else}_{S_2}\ C_2$	conditional
$C ::=$	Composite Statement
$\quad \textbf{nil}$	empty statement
$\quad I$	single statement
$\quad C; C$	sequencing

When making an observation $\textbf{observe}_b$, we make explicit the type b of the observed location. In the form $\textbf{if}\ l\ \textbf{then}_{S_1}\ C_1\ \textbf{else}_{S_2}\ C_2$, the sets S_1 and S_2 denote the local variables of the **then** branch and the **else** branch, respectively. These annotations simplify type checking and denotational semantics; they could easily be inferred.

The typing rules for Imp are standard. We consider Imp typing environments Σ to be a special case of Fun environments, where variables (locations) always map to base types. The judgment $\Sigma \vdash C : \Sigma'$ means that the composite statement C is well-typed in the initial context Σ, yielding additional bindings Σ'. Let $\Sigma \setminus S$ be the environment Σ restricted to only the locations in $\mathrm{dom}(\Sigma) \setminus S$. Below, we give some representative rules.

Part of the Type System for Imp:

$\Sigma ::= \varepsilon \mid \Sigma, l{:}b$ — Heap typings

(IMP ASSIGN)
$$\frac{\Sigma \vdash E : b \qquad l \notin \mathrm{dom}(\Sigma)}{\Sigma \vdash l \leftarrow E : \varepsilon, l{:}b}$$

(IMP SEQ)
$$\frac{\Sigma \vdash C_1 : \Sigma' \qquad \Sigma, \Sigma' \vdash C_2 : \Sigma''}{\Sigma \vdash C_1; C_2 : \Sigma', \Sigma''}$$

(IMP NIL)
$$\frac{\Sigma \vdash \diamond}{\Sigma \vdash \textbf{nil} : \varepsilon}$$

(IMP IF)
$$\frac{\Sigma \vdash l : \textbf{bool} \qquad \Sigma \vdash C_1 : \Sigma_1 \qquad \Sigma \vdash C_2 : \Sigma_2 \qquad \Sigma' = \Sigma_1 \setminus S_1 = \Sigma_2 \setminus S_2}{\Sigma \vdash \textbf{if}\ l\ \textbf{then}_{S_1}\ C_1\ \textbf{else}_{S_2}\ C_2 : \Sigma'}$$

4.2 Distribution Transformer Semantics of Imp

Similarly to Fun, a compound statement in Imp has a semantics as a distribution transformer generated from the set of combinators defined in Section 3. An Imp program does not return a value, but is solely a distribution transformer on states $\mathtt{S}\langle\Gamma\rangle$ (where Γ is a special case of Σ).

Interpretation of Statements: $\mathcal{I}[\![C]\!], \mathcal{I}[\![I]\!] : \mathtt{S}\langle\Sigma\rangle \rightsquigarrow \mathtt{S}\langle\Sigma'\rangle$

$\mathcal{I}[\![\textbf{nil}]\!] \triangleq \mathtt{pure\ id}$
$\mathcal{I}[\![l \leftarrow c]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, c)$
$\mathcal{I}[\![l \leftarrow l']\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, \mathtt{lookup}\ l'\ s)$
$\mathcal{I}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, (\mathtt{lookup}\ l_1\ s \otimes \mathtt{lookup}\ l_2\ s)))$
$\mathcal{I}[\![l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)]\!] \triangleq$
$(\mathtt{extend}\ \lambda s.D(\mathtt{lookup}\ l_1\ s, \ldots, \mathtt{lookup}\ l_n\ s)) \ggg (\mathtt{pure\ add}\ l)$

$\mathcal{I}[\![\textbf{observe}_b\ l]\!] \triangleq \mathtt{observe}\ \lambda s.\mathtt{lookup}\ l\ s$

$\mathcal{I}[\![\textbf{if}\ l\ \textbf{then}_{S_1}\ C_1\ \textbf{else}_{S_2}\ C_2]\!] \triangleq \mathtt{choose}\ \lambda s.\mathtt{if}\ (\mathtt{lookup}\ l\ s)$
$\mathtt{then}\ (T_1 \ggg \mathtt{pure}\ (\mathtt{drop}\ S_1))\ \mathtt{else}\ (T_2 \ggg \mathtt{pure}\ (\mathtt{drop}\ S_2))$

$\mathcal{I}[\![C_1; C_2]\!] \triangleq \mathcal{I}[\![C_1]\!] \ggg \mathcal{I}[\![C_2]\!]$

The main difference to the semantics of Fun is that Imp programs do not return values, and that local variables are dropped all at once at the exit of a **then** or **else** branch, but not elsewhere.

4.3 Translating from Fun to Imp

The translation from Fun to Imp is, for the most part, a straightforward compilation of functional code to imperative code. The only complication is that Imp is entirely unstructured: we can't use contiguous chunks of memory for tuple layout, which entails a little more management than is typically necessary. We work around this problem by mapping Fun's structured types to heap locations of base type in Imp with *patterns*, defined below.

Type/Pattern Agreement:

$p ::= l \mid (p, p)$ Patterns

(PAT LOC) $\dfrac{l : t \in \Sigma \quad l \neq ()}{\Sigma \vdash l : t}$ (PAT UNIT) $\dfrac{}{\Sigma \vdash () : \textbf{unit}}$ (PAT PAIR) $\dfrac{\Sigma \vdash p_1 : t_1 \qquad \Sigma \vdash p_2 : t_2}{\Sigma \vdash (p_1, p_2) : t_1 * t_2}$

Values of base type are represented by a single location; products are represented by a pattern for their corresponding components. We interpret heap typings Σ with a heap layout ρ, which is a partial function from Fun variables to patterns. Together, a typing Σ and a layout ρ correspond to a functional context Γ:

$$\Sigma \vdash \rho : \Gamma \iff \mathrm{dom}(\rho) = \mathrm{dom}(\Gamma) \wedge \forall x : t \in \Gamma.\Sigma \vdash \rho(x) : t$$

Let $\sim$ be the congruence closure on patterns of the total relation on locations. We write $p \leftarrow p'$ for piecewise assignment of $p \sim p'$. We say $p \in \Sigma$ if every location in p is in Σ. Finally, we silently thread a source of fresh locations through the following judgment.

Translation: $\rho \vdash M \Rightarrow C, p$

(TRANS VAR) $\dfrac{}{\rho \vdash x \Rightarrow \textbf{nil}, \rho(x)}$ (TRANS CONST) $\dfrac{c \neq ()}{\rho \vdash c \Rightarrow l \leftarrow c, l}$ (TRANS UNIT) $\dfrac{}{\rho \vdash () \Rightarrow \textbf{nil}, ()}$

(TRANS PAIR) $\dfrac{\rho \vdash V_1 \Rightarrow C_1, p_1 \qquad \rho \vdash V_2 \Rightarrow C_2, p_2}{\rho \vdash (V_1, V_2) \Rightarrow C_1; C_2; (p_1, p_2)}$ (TRANS PROJ1) $\dfrac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.1 \Rightarrow C, p_1}$

(TRANS IF) $\dfrac{\rho \vdash V_1 \Rightarrow C_1, l \qquad p_2 \sim p \text{ fresh} \qquad \rho \vdash M_2 \Rightarrow C_2, p_2 \qquad \rho \vdash M_3 \Rightarrow C_3, p_3}{\rho \vdash \textbf{if}\ V_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 \Rightarrow C_1; \textbf{if}\ l\ \textbf{then}\ C_2; p \leftarrow p_2\ \textbf{else}\ C_3; p \leftarrow p_3, p}$ (TRANS PROJ2) $\dfrac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.2 \Rightarrow C, p_2}$

(TRANS OBSERVE) $\dfrac{\rho \vdash V \Rightarrow C, p}{\rho \vdash \textbf{observe}\ V \Rightarrow C; \textbf{observe}_b\ p, ()}$ (TRANS SAMPLE) $\dfrac{\rho \vdash V \Rightarrow C, p}{\rho \vdash \textbf{sample}\ (D(V)) \Rightarrow C; l \overset{s}{\leftarrow} D(p), l}$

(TRANS LET) $\dfrac{\rho \vdash M_1 \Rightarrow C_1, p_1 \qquad \rho\{x \mapsto p_1\} \vdash M_2 \Rightarrow C_2, p_2}{\rho \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \Rightarrow C_1; C_2, p_2}$

In general, a Fun term M translates under a layout ρ to a series of commands C and a pattern p. The commands C mutate the global store so that the locations in p correspond to the value that M returns. The simplest example of this is in (TRANS CONST): the constant expression c translates to an Imp program that writes c into a fresh location l. The pattern that represents this return value is l itself. The (TRANS VAR) and (TRANS UNIT) rules are similar. In both rules, no commands are run. For variables, we look up the pattern in the layout ρ; for unit, we return the unit location. Translation of pairs (TRANS PAIR) builds each of the constituent values and constructs a new pair pattern.

More interesting are the projection operators. Consider (TRANS PROJ1); the second projection is translated similarly by (TRANS PROJ2). To find $V.1$, we run the commands to generate V, which we know must return a pair pattern (p_1, p_2). To extract the first element of this pair, we simply need to return p_1. Not only would it not be easy to isolate and run only the commands to generate the values that go in p_1, it would be incorrect to do so. For example, the Fun expressions constructing the second element of V may observe values, which has non-local effects.

The translation for conditionals (TRANS IF) is somewhat subtle. First, the conditional and branches are translated, and the commands to generate the result of conditional are run before the test itself. Next, a pattern p of fresh locations is used to hold the return value; using a shard output pattern allows us to avoid the ϕ nodes common in SSA compilers. Finally, we mentioned earlier that the **then** and **else** branches must be marked with the locations that are local to them. In this case, those locations are all of the locations written in the branch *except* the locations in the shared target pattern p.

We translate **observe** by running the commands to generate the value for V and then observing the pattern (TRANS OBSERVE). (Just as for Fun, we restrict observations to locations of base type—so p is a location l.) Sampling is translated similarly to observation (TRANS SAMPLE). By $D(p)$, we mean the flattening of p into a list of locations and passing it to the distribution constructor D.

Finally, we translate **let** statements with (TRANS LET) by simply running both expressions in sequence. Note that we translate M_2, the body of the let, with an extended layout, so that C_2 knows where to find the values written by C_1—in the pattern p_1.

PROPOSITION 4 *If* $\Gamma \vdash M : t$ *and* $\Sigma \vdash \rho : \Gamma$ *then there exists a fresh* Σ' *such that:* $\rho \vdash M \Rightarrow C, p$ *and* $\Sigma, \Sigma' \vdash p : t$ *and* $\Sigma \vdash C : \Sigma'$.

Let lift $\rho \triangleq \lambda s.\{\rho(x) \mapsto \mathcal{V}[\![x]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$ (where mapping is piecewise for the locations inside structured patterns) and restrict $\rho \triangleq \lambda s.\{x \mapsto \mathcal{V}[\![\rho(x)]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$. These translate to and from distributions on Fun variables ($\mathtt{S}\langle\Gamma\rangle$) to distributions on Imp locations ($\mathtt{S}\langle\Sigma\rangle$), respectively. We write $p\ s$ for the reconstruction of a p-structured value by looking up its locations in s.

THEOREM 2 $\Gamma \vdash M : t$ *and* $\Sigma \vdash \rho : \Gamma$ *and* $\rho \vdash M \Rightarrow C, p$ *then* $\mathcal{A}[\![M]\!] = \mathtt{pure}\ (\mathrm{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \mathtt{pure}\ (\lambda s.\ (\mathrm{restrict}\ \rho\ s, p\ s))$.

4.4 Factor Graphs

A factor graph (Kschischang et al. 2001) represents a joint probability distribution of a set of random variables as a collection of multiplicative factors. Factor graphs are an effective means of stating conditional independence properties between variables, and enable efficient algebraic inference techniques (Minka 2001; Winn and Bishop 2005) as well as sampling techniques (Koller and Friedman 2009, Chapter 12). We use factor graphs with *gates* (Minka

and Winn 2008) for modelling if-then-else clauses; this introduces second-order edges in the graph.

Factor Graphs:

$x, y, z, \ldots$	Variables
$G ::= \mathsf{new}\ \bar{x}\ \mathsf{in}\ \{e_1, \ldots, e_n\}$	Graph
$e ::=$	Edge
$\quad \mathsf{Equal}(x, y)$	Equality $(x = y)$
$\quad \mathsf{Constant}_c(x)$	Constant $(x = c)$
$\quad \mathsf{Binop}_{\otimes}(x, y, z)$	Binary operator $(x = y \otimes z)$
$\quad \mathsf{Sample}_D(x, y_1, \ldots, y_n)$	Sampling $(x \sim D(y_1, \ldots, y_n))$
$\quad \mathsf{Select}_n(x, v, y_1, \ldots, y_n)$	(De)Multiplexing $(x = y_v)$
$\quad \mathsf{Gate}(x, G_1, G_2)$	Gate (if x then G_1 else G_2)

In $\mathsf{new}\ \bar{x}\ \mathsf{in}\ \{e_1, \ldots, e_n\}$, the variables $\bar{x}$ are bound, and subject to α-renaming. We write $\mathrm{fv}(G)$ for the variables occurring free in G. If statements introduce gates, that are higher-order edges. All variables free in the branches of the if statement are also accessible in the graph containing the gate. Here we give a factor graph G_{E} corresponding to the epidemology example of Section 2.4 (where $B = \mathbf{Bernoulli}$).

Epidemiology Example, Take 2:

$$
\begin{array}{l}
G_{\mathrm{E}} = \{\mathsf{Constant}_{0.01}(p_d), \mathsf{Sample}_B(\mathsf{has\_disease}, p_d), \\
\mathsf{Gate}(\mathsf{has\_disease}, \\
\quad \mathsf{new}\ p_p\ \mathsf{in}\ \{\mathsf{Constant}_{0.8}(p_p), \mathsf{Sample}_B(\mathsf{positive\_result}, p_p)\}, \\
\quad \mathsf{new}\ p_n\ \mathsf{in}\ \{\mathsf{Constant}_{0.096}(p_n), \mathsf{Sample}_B(\mathsf{positive\_result}, p_n)\}), \\
\mathsf{Constant}_{\mathbf{true}}(\mathsf{positive\_result})\}
\end{array}
$$

A factor graph normally denotes a probability distribution. The probability (density) of an assignment of values to variables is equal to the product of all the factors, averaged over all assignments to local variables. Here, we give a more general semantics of factor graphs as distribution transformers; the input distribution corresponds to a prior factor over all variables that it mentions. Below, we use the Iverson brackets, where $[p]$ is 1 when p is true and 0 otherwise. We let $\delta(x = y) \triangleq \delta_0(x - y)$ when x, y denote real numbers, and $[x = y]$ otherwise.

Distribution Transformer Semantics: $\mathcal{P}[\![G]\!]^{\Sigma'}_{\Sigma} \in \mathsf{S}\langle\Sigma\rangle \rightsquigarrow \mathsf{S}\langle\Sigma, \Sigma'\rangle$

$$
\begin{array}{l}
\mathcal{P}[\![G]\!]^{\Sigma'}_{\Sigma}\ \mu\ A \triangleq \frac{1}{Z} \int_A \mathcal{P}[\![G]\!]\ s\, d(\mu \times \lambda)(s) \\
\qquad \text{where } Z = \int_{V_{\mathsf{S}\langle\Sigma,\Sigma'\rangle}} \mathcal{P}[\![G]\!]\ s\ d(\mu \times \lambda)(s) \\
\mathcal{P}[\![\mathsf{new}\ \overline{x : b}\ \mathsf{in}\ \{\bar{e}\}]\!]\ s \triangleq \int_{\times_j V_{b_j}} \prod_i (\mathcal{P}[\![e_i]\!]\ (s, \bar{y}))\, d\bar{y} \\
\mathcal{P}[\![\mathsf{Equal}(l, l')]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ l'\ s) \\
\mathcal{P}[\![\mathsf{Constant}_c(l)]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = c) \\
\mathcal{P}[\![\mathsf{Binop}_{\otimes}(l, w_1, w_2)]\!]\ s \triangleq \\
\quad \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ w_1\ s \otimes \mathtt{lookup}\ w_2\ s) \\
\mathcal{P}[\![\mathsf{Sample}_D(l, v_1, \ldots, v_n)]\!]\ s \triangleq \\
\quad D(\mathtt{lookup}\ v_1\ s, \ldots, \mathtt{lookup}\ v_n\ s)\ (\mathtt{lookup}\ l\ s) \\
\mathcal{P}[\![\mathsf{Select}_n(l, v, y_1, \ldots, y_n)]\!]\ s \triangleq \prod_i \delta(l = y_i)^{[v=i]} \\
\mathcal{P}[\![\mathsf{Gate}(v, G_1, G_2)]\!]\ s \triangleq (\mathcal{P}[\![G_1]\!]\ s)^{[\mathtt{lookup}\ v\ s]} (\mathcal{P}[\![G_2]\!]\ s)^{[\neg \mathtt{lookup}\ v\ s]}
\end{array}
$$

4.5 Factor Graph Semantics for Imp

An Imp statement can easily be given a semantics as a factor graph.

Factor Graph Interpretation: $\mathcal{G}[\![C]\!]G$

$$
\begin{array}{l}
\mathcal{G}[\![\mathbf{nil}]\!] \triangleq \varnothing \\
\mathcal{G}[\![l \leftarrow c]\!] \triangleq \mathsf{Constant}_c(l) \\
\mathcal{G}[\![l \leftarrow l']\!] \triangleq \mathsf{Equal}(l, l') \\
\mathcal{G}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \mathsf{Binop}_{\otimes}(l, l_1, l_2) \\
\mathcal{G}[\![l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)]\!] \triangleq \mathsf{Sample}_D(l, l_1, \ldots, l_n) \\
\mathcal{G}[\![\mathbf{observe}_b\ l]\!] \triangleq \mathsf{Constant}_{0_b}(l) \\
\mathcal{G}[\![\mathsf{weight}\ r]\!] \triangleq \mathsf{Weight}(r) \\
\mathcal{G}[\![\mathbf{if}\ l\ \mathbf{then}_{S_1}\ C_1\ \mathbf{else}_{S_2}\ C_2]\!] \triangleq \mathsf{Gate}(l, \mathsf{new}\ S_1\ \mathsf{in}\ G_1, \mathsf{new}\ S_2\ \mathsf{in}\ G_2) \\
\mathcal{G}[\![C_1; C_2]\!] \triangleq \mathcal{G}[\![C_1]\!] \cup \mathcal{G}[\![C_2]\!]
\end{array}
$$

THEOREM 3 *If* $\Sigma \vdash C : \Sigma'$ *and* $\mu \in \mathtt{D}\langle\mathsf{S}\langle\Sigma\rangle\rangle$ *is compatible with* $\mathcal{I}[\![C]\!]$ *then* $\mathcal{I}[\![C]\!]\ \mu\ A = \mathcal{P}[\![\mathcal{G}[\![C]\!]]\!]\ \mu\ A$ *for all measurable* $A \subseteq \mathcal{T}[\![\mathsf{S}\langle\Sigma, \Sigma'\rangle]\!]$.

5. Adding Arrays and Comprehensions

To be useful for machine learning, our language must support large datasets. To this end, we extend Fun and Imp with arrays and comprehensions. We offer a few examples, after which we present the formal semantics.

5.1 Comprehensions

Earlier, we tried to estimate the skill levels of three competitors in head-to-head games. Using comprehensions, we can model skill levels for an arbitrary number of players:

TrueSkill:

```
let trueskill (players:int[]) (results:(bool*int*int)[]) =
    let skills = [for p in players → sample (Gaussian(10.0,20.0))]
    for (w,p1,p2) in results do
        let perf1 = sample (Gaussian(skills.[p1], 1.0))
        let perf2 = sample (Gaussian(skills.[p2], 1.0))
        if w // win?
        then observe (perf1 > perf2) // first player won
        else observe (perf1 − perf2) // draw
    ps
```

First, we create a prior distribution for each player: we assume that skill is normally distributed around 10.0, with variance 20.0. Then we look at each of the results—this is the comprehension. The results of the head-to-head matches are a list of triples: a boolean and two indices. If the boolean is true, then the first index represents the winner and the second represents the loser. If the boolean is false, then the match was a draw between the two players. The probabilistic program walks over the results, and observes that either the first player's performance—normally distributed around their skill level—was greater than the second's performance, or that the two players' performances were equal. Returning ps after these observations allows us to inspect the posterior distributions. Our original example can be modelled with players $= [0; 1; 2]$ (IDs for Alice, Bob, and Cyd, respectively) and results $= [\mathbf{true}, 0, 1; \mathbf{true}, 1, 2; \mathbf{true}, 0, 2]$.

We can generalize the category parameter learning if we keep our data in arrays, as follows:

Bayesian Inference over Arrays:

```
let classify (objs:int[]) (weights:(int*real)[]) weightMean
      weightVariance =
    let priors = [for oid in objs → sample (Gaussian(weightMean,
          weightVariance))]
    for (oid,seen) in weights do
        let meas = sample (Gaussian(priors.[oid],1.0))
        observe (meas − seen)
    priors
let objs = (* ... *)
let weights = (* ... *)
```

The function classify is a probabilistic program for the learning version of a naive Bayesian classifier. Each category of objects—modelled by the array objs—is given a normally distributed prior

on weight of objects in that category; these are held in the priors array. Then, for each measurement seen of an object of category oid in the weights array, we observe that seen is normally distributed according to the prior for that category of object. We then return the posterior distributions, which have been appropriately modified by the observed weights. We can train using this model by running a command such as classify objs weights 20.0 5.0.

5.2 Formalizing Arrays and Comprehensions

We define primitives for arrays in Fun and Imp, and give their interpretation as distribution transformers and factor graphs.

In Fun, there are three array operations: constant-size arrays, indexing, and array comprehension. First, let $\mathcal{R}$ be a set of *ranges* r. Ranges allow us to differentiate arrays of different sizes. We assign sizes to ranges using the function $|\cdot| : \mathcal{R} \to \mathbb{Z}^+$. In the metalanguage, arrays over range r correspond to tuples of length r; array indexing is modulo the size of the array.

Extended Syntax of Fun:

$t ::= \ldots \mid t[r]$	Types
$M,N ::= \ldots \mid$	Expression
$\quad [V_1;\ldots;V_n]$	array literal
$\quad V_1.[V_2]$	indexing
$\quad [\textbf{for } x \textbf{ in}_r V \to M]$	comprehensions

First, we add arrays as a type: $t[r]$ is an array elements of type t over the range r. Indexing, $V_1.[V_2]$, extracts elements out of a collection. Comprehensions map over arrays, producing a new array where each element is determined by evaluating E with each element of V bound to x in turn. We attach the range to a comprehension so that the distribution transformer semantics can be given simply; this argument can be inferred easily, and need not be written by the programmer. We elide the range in our code examples. In this formalism, we don't distinguish comprehensions that produce values—like the one that produces ps—and those that do not— like the one that observes player performances according to results. For efficiency reasons, our implementation does distinguish these two uses. In some of the code examples, we write **for** x **in** V **do** M to mean $[\textbf{for } x \textbf{ in}_r V \to M]$— we do this only when M has type **unit** and we do not care about the resulting array of unit values.

To simplify the distribution transformer semantics, we require that the elements of constant arrays, the operands of indexing, and the subject of a comprehension are values. When non-value expressions occur in these positions, we take them to mean the standard **let** expansion.

Extended typing rules for Fun expressions: $\Gamma \vdash M : t$

(Fun Array)
$$\frac{\Gamma \vdash M_i : t \quad \forall i \in 1..n}{\Gamma \vdash [M_1;\ldots;M_n] : t[r_n]}$$

(Fun Index)
$$\frac{\Gamma \vdash M_1 : t[r] \qquad \Gamma \vdash M_2 : \textbf{int}}{\Gamma \vdash M_1[M_2] : t}$$

(Fun For)
$$\frac{\Gamma \vdash M_1 : t[r] \qquad \Gamma, x : t \vdash M_2 : t}{\Gamma \vdash [\textbf{for } x \textbf{ in}_r M_1 \to M_2] : t[r]}$$

The static semantics of these new constructs is straightforward. Constant arrays of size n are given collection types with a "concrete" range r_n, unique for each n. Indexing requires that the operands be an array and an integer, respectively. Array types include ranges to aid compilation, but are not intend to prevent out-of-bounds accesses statically. Comprehensions require that the source expression M_1 be well typed as an array, and that the yielding expression M_2 be well typed assuming a suitable type for x.

The distribution transformer semantics models arrays over a range r as tuples of length $|r|$. Comprehensions are treated as their unrolling, using a series of temporary variables to hold on to intermediate elements of the output array.

Extended Distribution Transformer Semantics of Fun:

$$\mathcal{T}[\![t[r]]\!] = (V_t^{|r|}, \sigma_{V_t^{|r|}}(\{m_1,\ldots,m_{|r|} \mid m_i \in \mathcal{M}_t\}))$$

$$\mathcal{A}[\![[V_1;\ldots;V_n]]\!] = \texttt{pure}\ (\lambda s.(s, [\mathcal{V}[\![V_1]\!]\ s;\ldots;\mathcal{V}[\![V_n]\!]\ s]))$$
$$\mathcal{A}[\![V_1[V_2]]\!] = \texttt{pure}\ (\lambda s.(s, (\mathcal{V}[\![V_1]\!]\ s)[\mathcal{V}[\![V_2]\!]\ s]))$$
$$\mathcal{A}[\![\textbf{for } x \textbf{ in}_r V \to M]\!] =$$
$$\quad \mathcal{A}[\![V]\!] \ggg \texttt{pure}\ (\texttt{add}\ x) \ggg$$
$$\quad \mathcal{A}[\![M\left\{{}^{x[0]}/_{x}\right\}]\!] \ggg \texttt{pure}\ (\texttt{add}\ y_1) \ggg \cdots \ggg$$
$$\quad \mathcal{A}[\![M\left\{{}^{x[|r|-1]}/_{x}\right\}]\!] \ggg \texttt{pure}\ (\texttt{add}\ y_{|r|}) \ggg$$
$$\quad \mathcal{A}[\![[y_1;\ldots;y_{|r|}]]\!] \ggg$$
$$\quad \texttt{pure}\ \lambda(s,v).((\texttt{drop}\ \{x,y_1,\ldots,y_{|r|}\}\ s),v)$$

We require even fewer new primitives to support collections in Imp. We extend the assignment form to allow for array indexing and iteration over ranges.

Extended Syntax of Imp:

$lhs ::= l \mid lhs[E] \mid lhs[r]$	Left-hand sides, subset of E
$E ::= c \mid l \otimes l \mid lhs$	Expressions
$I ::= \ldots \mid$	Statements
$\quad lhs \leftarrow E$	assignment
$\quad \textbf{for } r \textbf{ do } C$	iteration over ranges

Adapting the static semantics of Imp requires slightly more work. First, we keep track of which ranges are currently being iterated over: it is nonsense to index an array by an inactive range. Next, heap typings distinguish locations of base type and locations of array type. When we say $l : b[r_1][\ldots][r_n]$, we mean an array of arrays (of arrays...) of values of type b. Earlier, we wrote $p \leftarrow p'$ to mean the piecewise assignment from p' into a congruent pattern p. We write $p[r] \leftarrow p'[r]$ to mean **for** r **do** $p[r] \leftarrow p'[r]$, i.e., the index-wise assignment p' into an array over the same range.

Extended Typing Rules for Imp:

$\Sigma ::= \varepsilon \mid \Sigma, l : b \mid \Sigma, l : b[r_1][\ldots][r_n]$	Heap typings
$R ::= \varepsilon \mid R, r$	Range tracking

(Imp Index)
$$\frac{\Sigma; R \vdash E_1 : t[r] \qquad \Sigma; R \vdash E_2 : \textbf{int}}{\Sigma; R \vdash E_1[E_2] : t}$$

(Imp Range)
$$\frac{\Sigma; R \vdash E : t[r] \qquad r \in R}{\Sigma; R \vdash E[r] : t}$$

(Imp Assign)
$$\frac{\Sigma; R \vdash lhs : b \qquad \Sigma; R \vdash E : b}{\Sigma \vdash lhs \leftarrow E}$$

(Imp For)
$$\frac{\Sigma; R, r \vdash C}{\Sigma; R \vdash \textbf{for } r \textbf{ do } C}$$

We omit the R in most judgments, where we simply threaded through: it is meaningful only for (Imp Range) and (Imp For).

Just as in Fun, the semantics does not need to change much. We interpret comprehensions as their unrolling.

Distribution Transformer Semantics of Imp Collections:

$$\mathcal{F}[\![E_1[E_2]]\!]\ s \triangleq (\mathcal{F}[\![E_1]\!]\ s)[\mathcal{F}[\![E_2]\!]\ s]$$

$$\mathcal{A}[\![l \leftarrow E]\!] \triangleq \texttt{pure}\ \lambda s.\texttt{add}\ l\ (s, \mathcal{F}[\![E]\!]\ s)$$
$$\mathcal{A}[\![lhs[E_1] \leftarrow E_2]\!] \triangleq \texttt{pure}\ \lambda s.\texttt{add}\ \mathcal{F}[\![lhs[E_1]]\!]\ \ (s, \mathcal{F}[\![E_2]\!]\ s)$$
$$\mathcal{A}[\![\textbf{for } r \textbf{ do } C]\!] \triangleq \mathcal{A}[\![C\left\{{}^{0}/_{r}\right\}; \ldots; C\left\{{}^{|r|-1}/_{r}\right\}]\!]$$

To prove that the distribution transformer semantics of the extended Fun and Imp correspond, we must extend some of our other relations. The judgment for pattern and type agreement must correctly correlate collection types and patterns. We define $(p_1, p_2)[r]$ as $(p_1[r], p_2[r])$.

Pattern/Type Agreement:

$$\frac{l : t \in \Sigma}{\Sigma \vdash l : t}\ (\textsc{Pat Loc}) \qquad \frac{\Sigma \vdash p_1 : t_1 \quad \Sigma \vdash p_2 : t_2}{\Sigma \vdash (p_1, p_2) : t_1 * t_2}\ (\textsc{Pat Pair})$$

$$\frac{\Sigma \vdash p_1 : t_1[r_1][\ldots][r_n] \quad \Sigma \vdash p_2 : t_2[r_1][\ldots][r_n]}{\Sigma \vdash (p_1, p_2) : (t_1 * t_2)[r_1][\ldots][r_n]}\ (\textsc{Pat Array Pair})$$

The (Pat Loc) rule covers locations of both base and array types. The (Pat Pair) rule is unchanged, but is repeated here for clarity. The (Pat Array Pair) rule explains how we represent arrays of tuples in Fun as tuples of arrays in Imp.

New Translation Rules:

$$\frac{\rho \vdash M_i \Rightarrow C_i, p_i \qquad p_i[r_n] \sim p \text{ fresh}}{\rho \vdash [V_1; \ldots; V_n] \Rightarrow C_1; \ldots; C_n; p[i] \leftarrow p_i, p}\ (\textsc{Trans Array})$$

$$\frac{\rho \vdash V_1 \Rightarrow C_1, p_1 \qquad \rho \vdash V_2 \Rightarrow C_2, l_2 \qquad p_1 \sim p \text{ fresh}}{\rho \vdash V_1[V_2] \Rightarrow C_1; C_2; p \leftarrow p_1[l_2], p}\ (\textsc{Trans Index})$$

$$\frac{\rho \vdash M_1 \Rightarrow C_1, p_1 \qquad M_1 : t[r] \qquad \rho\{x \mapsto p_1[r]\} \vdash M_2 \Rightarrow C_2, p_2 \qquad p[r] \sim p' \, fresh}{\rho \vdash [\textbf{for } x \textbf{ in}_r \, M_1 \rightarrow M_2] \Rightarrow C_1; \textbf{for } r \textbf{ do } \{C_2; p'[r] \leftarrow p\}, p'}\ (\textsc{Trans For})$$

We note that the sizes of ranges are never needed in this translation; compilation is not data dependent. Given these definitions, we restate Proposition 4 and Theorem 2:

Proposition 5 *If* $\Gamma \vdash M : t$ *and* $\Sigma \vdash \rho : \Gamma$ *then there exists a fresh* Σ' *such that:* $\rho \vdash M \Rightarrow C, p$ *and* $\Sigma, \Sigma' \vdash p : t$ *and* $\Sigma \vdash C : \Sigma'$.

Theorem 4 $\Gamma \vdash M : t$ *and* $\Sigma \vdash \rho : \Gamma$ *and* $\rho \vdash M \Rightarrow C, p$ *then* $\mathcal{A}[\![M]\!] = \texttt{pure}\ (\text{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \texttt{pure}\ (\lambda s.\ (\text{restrict}\ \rho\ s, p\ s))$.

The factor graph semantics (omitted for lack of space) is also based on the unrolling of loops, implementing indexing using the factor $\mathsf{Select}_n(l, v, y_1, \ldots, y_n)$ previously defined.

6. Implementation Experience

We implemented a compiler from Fun to Imp in F#. We wrote two backends for Imp: an exact inference algorithm for discrete distributions, obtained by treating our distribution transformers as F# code and an approximating infernce algorithm for continuous distributions, using Infer.NET (Minka et al. 2009). The translation of Section 4 formalizes our translation of Fun to Imp. Translating Imp to Infer.NET is relatively straightforward, and amounts to a syntax-directed series of calls to Infer.NET's object-oriented API.

The frontend of our compiler takes (a subset of) actual F# code as its input. To do so, we make use of F#'s *reflected definitions*, which allow programmatic access to ASTs. This implementation strategy is advantageous in several ways. First, there is no need to design new syntax, or even write a parser. Second, all inputs to our compiler are typed ASTs of well typed F# programs. Third, a single file can contain both ordinary F# code as well as reflected definitions. This allows, in particular, a single module to read in data from, for example, a database, process it, and register it with our compiler. Reflected definitions later in the same file are typed with respect to these registered definitions and then run in Infer.NET with the pre-processed data; we discuss this idea more below.

We offer some statistics on a few of the examples we've implemented. The lines of code number includes F# code that loads and processes data from disk before loading it into Infer.NET. The times are based on an average of three runs. All of the runs are on a four-core machine with 4GB of RAM. The Naive Bayes program is the naive Bayesian classifier of the earlier examples. The Mixture model is another clustering/classification model. TrueSkill was described earlier, and adPredictor is described below. Of the two long-running examples, time is spent mostly loading and processing data from disk and running inference in Infer.NET. TrueSkill spends the majority of its time—64%—during inference in Infer.NET. Much more of adPredictor's time is spent in pre-processing: 58%, with 40% of its time in inference. The time spent in our compiler is negligible, never more than a few hundred milliseconds.

Summary of our Basic Test Suite:

	LOC	Observations	Variables	Time
Naive Bayes	28	9	3	<1s
Mixture	33	3	3	<1s
TrueSkill	68	15,664	84	6s
adPredictor	78	300,752	299,594	3m30s

In summary, our implementation strategy allowed us to build an effective prototype quickly and easily: the entire compiler is only 2079 lines of F#; the Infer.NET backend is 600 lines; the discrete backend is 252 lines. Our implementation, however, is only a prototype, and has limitations. For one, Infer.NET supports a limited set of operations on specific combinations of stochastic and deterministic arguments. It also limits the dimensionality of arrays. It would be useful in the future to have an enhanced type system able to detect errors arising from illegal combinations of operators in Infer.NET. Our discrete backend is limited to discrete distributions over finite sets. The reflected definition facility is somewhat limited in F#. In the example below, a call to Array.toList is required because F# does not reflect definitions that contain comprehensions over arrays—only lists. (The F# to Fun compiler discards this extra call as a no-op, so there is no runtime overhead.)

In following two subsections, we describe in greater detail two probabilistic programs based on the adPredictor system (Graepel et al. 2010): one in Fun and one in PQL (Van Gael et al. 2010).

6.1 adPredictor in Fun

adPredictor, a component of the Bing search engine, estimates the click-through rates for particular users on advertisements. We first describe a probabilistic program for our system that models (a small part of) adPredictor, after which we describe a similar model in PQL. In the following models, we—without loss of generality—use only two features to make our prediction: the advertiser's listing and the phrase used for searching. In the real system, many more undisclosed features are used for prediction.

The following is an F# fragment of our adPredictor model.

adPredictor:

```
let read_lines filename count line = (* ... *)

[<RegisterArray>]
let imps = (* ... *)

[<ReflectedDefinition>]
let probit b x =
    let y = sample (Gaussian(x,1.0))
    if b then constrain (y > 0.0) else constrain (y < 0.0)

[<ReflectedDefinition>]
let ad_predictor (listings:int[]) (phrases:int[]) impressions =
    let lws = [for l in listings →sample (Gaussian(0.0,0.33))]
    let pws = [for p in phrases →sample (Gaussian(0.0,0.33))]
    for (clicked,lid,pid) in Array.toList impressions do
        probit clicked (lws.[lid] + pws.[pid])
    lws,pws
```

The read_lines function loads data in from a file on disk. The data are formatted as newline-separated records of comma-separated values. There are three important values in each record: a field that is 1 if the given impression lead to a click, and a 0 otherwise; a field that is the database ID of the listing shown; a field that is the part of the search phrase that led to the selection of the listing. We preprocess the data in three ways, which are elided in the code above. First, we convert the 1/0-valued boolean to a **true/false**-valued boolean. Second, we normalize the listing IDs so that they begin at 0, i.e., so that we can use them as array indices. Third, we collect unique phrases and assign them fresh, 0-based IDs. We define imps—a list of advertising impressions (a listing ID and a phrase ID) and whether or not the ad was clicked—in terms of this processed data. The [<RegisterArray>] attribute on the definition of imps instructs the compiler to load the result of this F# expression as a constant (also named imps) into Infer.NET. Finally, ad_predictor defines the model. We use the [<ReflectedDefinition>] attribute on ad_predictor to mark it as a probabilistic program, which should be compiled and sent to Infer.NET. Supposing we've stored the collated listing and phrase IDs in ls and ps, respectively; we can train on the impressions by calling ad_predictor ls ps imps.

6.2 adPredictor in PQL

PQL is one of several recent systems aimed at machine learning at scale (Malewicz et al. 2009; Yu et al. 2009). Given that machine learning applications are often driven by large relational datasets, the motivation for PQL is to put Bayesian models directly into the database. PQL's compiler works by transforming the relational, SQL-esque syntax into an approximative algorithm to be run on a cluster using DryadLinq. DryadLinq distributes the excecution of the algorithm over the cluster.

From a programmer's perspective, PQL models work by augmenting existing tables in a database with various factors.

adPredictor in PQL:

```
-- Add weights for all the features.
Listings = AUGMENT DB..Listings ADD Weight NORMAL;
Phrases = AUGMENT DB..Phrases ADD Weight NORMAL;

-- Add a prior distribution for all the features.
FACTOR Normal(x.Weight, 0.0, 0.33) FROM Listings x;
FACTOR Normal(x.Weight, 0.0, 0.33) FROM Phrases x;

-- Connect the features with the observed clicks
FACTOR Probit(imp.IsClicked, l.Weight, p.Weight)
FROM DB..Impressions imp
JOIN Listings l ON l.Listings = imp.Listing
JOIN Phrases p ON p.Phrase = imp.Phrase;
```

First, every listing and search phrase is given a prior distribution (normally distributed around 0.0, with variance 0.33). The model then joins the table of impressions, `Impressions`, with the listing and search phrase tables. The probit factor here has the same effect as the probit function above: if the given ad impression was clicked, then an observation boosts the weights of the given phrases and listings; if not, then the weights are pushed down.

The DryadLinq based implementation of PQL can process a slightly larger—but still small—dataset than we used for our Fun implementation. On a dataset with 10,343,631 ad impressions (ergo, observations), 241,710 listings and 67,404 phrases (309,114 variables), a ten-machine cluster took an average of 5 minutes 13 seconds. (Our system does not terminate on this dataset: it runs out of memory.) This is a comparable performance to the by-hand implementation of the real adPredictor system. Still, the startup cost of the DryadLinq implementation is about 4 minutes 30 seconds.

6.3 Comparing Fun and PQL

PQL scales better than our approach, though our language has several upsides. First, it is easier to compose programs in our setting. For example, we defined probit as a (reflected) function; in PQL, `Probit` is a built-in construct. Second, we can give a firm semantics to our language. Third, PQL is an experimental DSL, while our approach reuses the syntax, semantics, and implementation of an existing, relatively mature language, F#. Fourth, PQL weds itself to the database paradigm, while our approach to pre-processing allows programmers to learn from multiple data sources with a single model. Differences aside, we see the similarity as validating our approach: our principled semantics copes with a DSL serving the needs of machine learning practitioners.

In future work, we intend to translate PQL into Fun, possibly extended with features to support PQL joins, so as to extend our distribution transformer semantics to PQL. We have built an initial translator that yields inference times comparable to hand-written translations.

7. Related Work

To the best of our knowledge, this paper introduces the first rigorous measure-theoretic semantics and the first deterministic factor graph semantics for a probabilistic language with observation.

We are aware of only two prior probabilistic languages with operations like our **observe**. Csoft allows **observe** on zero probability events, but its semantics has not previously been formalized. A language by Kiselyov and Shan (2009) uses an explicit `fail` statement, which is equivalent to **observe false**, but cannot be used for conditioning on zero probability events. They implement a programming framework in OCaml, using explicit manipulation of discrete probability distributions as lists.

There is a long history of formal semantics for probabilistic languages with sampling primitives. Several of these languages implement a limited form of observations as conditional sampling; this entails using a loop around the sampling operation, a post-processing step or a conditional sampling primitive. Probabilistic LCF (Saheb-Djahromi 1978) augments the core functional language LCF with weighted binary choice, for discrete distributions.

Jones and Plotkin (1989) investigate the probability monad, and apply it to languages with discrete probabilistic choice. Ramsey and Pfeffer (2002) embed the probability monad within Haskell, with a measure-theoretic semantics; the do not consider observations.

Koller et al. (1997) pioneered the idea of representing a probability distribution using a functional program with discrete random choice, and proposed an inference algorithm for Bayesian networks and stochastic context-free grammars. Observations happen outside their language, by returning the three distributions $P(A \wedge B), P(A \wedge \neg B), P(\neg A)$ which can be used to compute $P(B \mid A)$.

Park et al. (2005) propose $\lambda_\circ$, the first probabilistic language with formal semantics applied to real problems involving continuous distributions. The formal basis is sampling functions, which uniformly supports both discrete and continuous probability distributions, and inference is by Monte Carlo methods. The calculus $\lambda_\circ$ does not include observations, but enables conditional sampling via fixpoints and rejection.

FACTORIE (McCallum et al. 2009) is a Scala library for explicitly constructing factor graphs. Although there are many Bayesian modelling languages, Csoft is the only previous language implemented by a compiling to factor graphs. We describe some of these other languages. Church (Goodman et al. 2008) is a probabilistic form of the untyped functional language Scheme, equipped with conditional sampling and a mechanism of stochastic memoization. Queries are implemented using Monte Carlo techniques. Blaise (Bonawitz 2008) supports the compositional construction of so-

phisticated probabilistic models, and decouples the choice of inference algorithm from the specification of the distribution. Ntzoufras (2009) describes the popular BUGS language, which explicitly describes distributions suitable for Monte Carlo analysis.

Probabilistic languages with formal semantics find application in many areas apart from machine learning, including databases (Dalvi et al. 2009), model checking (Kwiatkowska et al. 2006), differential privacy (McSherry 2009; Reed and Pierce 2010), information flow (Lowe 2002), and cryptography (Abadi and Rogaway 2002). The syntax of Imp is modelled on the probabilistic imperative language of Barthe et al. (2009).

Some recent program analyses rely on probabilistic inference on factor graphs (Kremenek et al. 2006; Livshits et al. 2009).

8. Conclusion

Our direct contribution is a rigorous semantics for a probabilistic programming language that also has an equivalent factor graph semantics. More importantly, the implication of our work for the machine learning community is that probabilistic programs can be written directly within an existing declarative language (Fun—a subset of F#), linked by comprehensions to large datasets, and compiled down to lower level Bayesian inference engines.

For the programming language community, our new semantics suggests some novel directions for research. What other primitives are possible—non-generative models, inspection of distributions, on-line inference on data streams? Can we provide a semantics for other data structures, especially of probabilistically varying size? Can our semantics be extended to higher-order programs? Can we verify the transformations machine learning compilers such as Infer.NET compiler for Csoft? Are there type systems for avoiding zero probability exceptions?

Acknowledgements We gratefully acknowledge discussions with Ralf Herbrich, Tom Minka, and John Winn. Neel Krishnaswami, Nikhil Swamy, and Dimitrios Vytiniotis provided helpful comments on drafts.

References

M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2): 103–127, 2002.

G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.

P. Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.

K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, MIT, 2008. Available as Technical Report MIT-CSAIL-TR-2008-044.

N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

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

T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale Bayesian click-through rate prediction for sponsored search advertising in Microsoft's Bing search engine. In *International Conference on Machine Learning*, 2010.

R. Herbrich, T. Minka, and T. Graepel. Trueskill(tm): A Bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, 2007.

E. T. Jaynes. *Probability Theory: The Logic of Science*, chapter 15.7 The Borel-Kolmogorov paradox, pages 467–470. CUP, 2003.

C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, pages 186–195. IEEE Computer Society, 1989.

O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *UAI*, 2009.

D. Koller and N. Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.

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

T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176. USENIX Association, 2006.

F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47 (2):498–519, 2001.

M. Z. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. *ENTCS*, 153(2):5–31, 2006.

V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, pages 75–86. ACM, 2009.

G. Lowe. Quantifying information flow. In *CSFW*, pages 18–31. IEEE Computer Society, 2002.

G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SPAA*, page 48. ACM, 2009.

A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs, 2009. Poster at 23rd Annual Conference on Neural Information Processing Systems (NIPS).

F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference*, pages 19–30. ACM, 2009.

E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of SIGMOD*, 2007.

T. Minka and J. M. Winn. Gates. In *NIPS*, 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`.

T. P. Minka. Expectation Propagation for approximate Bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann, 2001.

I. Ntzoufras. *Bayesian Modeling Using WinBUGS*. Wiley, 2009.

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

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

J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. Unpublished draft, 2010.

J. S. Rosenthal. *A First Look at Rigorous Probability Theory*. World Scientific, 2nd edition, 2006.

N. Saheb-Djahromi. Probabilistic LCF. In *MFCS*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.

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

D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. *SIGPLAN Not.*, 42(9):29–40, 2007b. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291159.

J. Van Gael, T. Graepel, R. Herbrich, et al. Probabilistic query language. In preparation, 2010.

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.

J. M. Winn and C. M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan. Some sample programs written in DryadLINQ. Technical Report MSR–TR–2009–182, Microsoft Research, 2009.

E. S. Yudkowsky. An intuitive explanation of Bayesian reasoning, 2003. Available at `http://yudkowsky.net/rational/bayes`.