Measure Transformer Semantics for Bayesian Machine Learning

Johannes Borgström[1], Andrew D. Gordon[1], Michael Greenberg[2], James Margetson[1], and Jurgen Van Gael[1]

[1] Microsoft Research
[2] University of Pennsylvania

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 combinators for measure transformers, based on theorems in measure theory, and use these to give a rigorous semantics to our core calculus. The original features of our semantics include its support for discrete, continuous, and hybrid measures, and, in particular, for observations of zero-probability events. We compile our core language to a small imperative language that has a straightforward semantics via factor graphs, data structures that enable many efficient inference algorithms. We use an existing inference engine for efficient approximate inference of posterior marginal distributions, treating thousands of observations per second for large instances of realistic models.

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. The theme of this paper is the idea of writing Bayesian models as probabilistic programs, which was pioneered by Koller et al. [16] and is recently gaining in popularity [31, 30, 9, 4, 14]. In particular, we draw inspiration from Csoft [37], an imperative language with an informal probabilistic semantics. Csoft is the native language of Infer.NET [25], a software library for Bayesian reasoning. A compiler turns Csoft programs into factor graphs [18], data structures that support efficient inference algorithms [15]. 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 [11], 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. In a Bayesian setting, we represent our uncertain knowledge of the skills as continuous probability distributions. The following probabilistic expression models the situation by generating probability distributions for the players' skills, given three played games (observations).

```
// prior distributions, the hypothesis
let skill() = random (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = random (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 goes as follows. We sample the skills of the three players from the *prior distribution* **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 centred 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.

A classic computational method to learn the posterior distribution of each of the skills is by Monte Carlo sampling [21]. We run the expression many times, but keep 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, 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.

Deterministic algorithms based on factor graphs [18, 15] are an efficient alternative to Monte Carlo sampling. To the best of our knowledge, all prior inference techniques for probabilistic languages, apart from Csoft and recent versions of IBAL [32], are based on nondeterministic inference using some form of Monte Carlo sampling. The benefit of using factor graphs in Csoft is to support deterministic but approximative inference algorithms, which are known to be significantly more efficient than sampling methods, where applicable.

Observations with zero probability arise commonly in Bayesian models. For example, in the model above, a drawn game would be modelled as the performance of two players being observed to be equal. Since the performances 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 in a Monte Carlo simulation. (To use Monte Carlo methods, one must instead write that the absolute difference between two drawn performances is less than some small ε.) However, our semantics based on measure theory makes sense of such observations, and corresponds to inference as achieved by algorithms on factor graphs.

Plan of the Paper We propose Fun:

- Fun is a functional language for Bayesian models with primitives for probabilistic sampling and observations (Section 2).
- Fun has a rigorous probabilistic semantics as measure transformers (Section 3).
- Fun has an efficient implementation: our system compiles Fun to Imp (Section 4), a subset of Csoft, and then relies on Infer.NET (Section 5).

Our main contribution is a framework for finite measure transformer semantics, which supports discrete measures, continuous measures, and mixtures of the two, and also supports observations of zero probability events.

As a substantial application, we supply measure transformer semantics for Fun, Imp, and factor graphs, and use the semantics to verify the translations in our compiler. Theorem 2 establishes the correctness of the translation from Fun to Imp and the factor graph semantics of Imp.

We designed Fun to be a subset of the F# dialect of ML [36], for implementation convenience: F# reflection allows easy access to the abstract syntax of a program. All the examples in the paper have been executed with our system, described in Section 5.

We end the paper with a description of related work (Section 6) and some concluding remarks (Section 7). A companion technical report [5] includes: detailed proofs; extensions of Fun, Imp, and our factor graph notations with array types suitable for inference on large datasets; listings of examples including versions of large-scale algorithms; and a description, including performance numbers, of our practical implementation of a compiler from Fun to Imp, and a backend based on Infer.NET.

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. We have arithmetic and Boolean operations on base types.

Types, Constant Data, and Zero Elements:

$a, b ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real}$ Base types

$t ::= \mathbf{unit} \mid b \mid (t_1 * t_2)$ Compound types

$\mathrm{ty}(()) = \mathbf{unit} \quad \mathrm{ty}(\mathbf{true}) = \mathrm{ty}(\mathbf{false}) = \mathbf{bool} \quad \mathrm{ty}(n) = \mathbf{int} \quad \mathrm{ty}(r) = \mathbf{real}$

$0_{\mathbf{bool}} = \mathbf{true} \qquad 0_{\mathbf{int}} = 0 \qquad 0_{\mathbf{real}} = 0.0$

Signatures of Arithmetic and Logical Operators: $\otimes : b_1, b_2 \to b_3$

$\&\&, ||, = : \textbf{bool}, \textbf{bool} \to \textbf{bool} \qquad >, = : \textbf{int}, \textbf{int} \to \textbf{bool}$

$+, -, * : \textbf{int}, \textbf{int} \to \textbf{int} \qquad > : \textbf{real}, \textbf{real} \to \textbf{bool} \qquad +, -, * : \textbf{real}, \textbf{real} \to \textbf{real}$

We have several standard probability distributions as primitive: $D : t \to u$ takes parameters in t and yields a random value in u.

Signatures of Distributions: $D : (x_1 : b_1 * \cdots * x_n : b_n) \to b$

$\textbf{Bernoulli} : (\mathsf{success} : \textbf{real}) \to \textbf{bool}$
$\textbf{Binomial} : (\mathsf{trials} : \textbf{int} * \mathsf{success} : \textbf{real}) \to \textbf{int}$
$\textbf{Poisson} : (\mathsf{rate} : \textbf{real}) \to \textbf{int}$
$\textbf{DiscreteUniform} : (\mathsf{max} : \textbf{int}) \to \textbf{int}$
$\textbf{Gaussian} : (\mathsf{mean} : \textbf{real} * \mathsf{variance} : \textbf{real}) \to \textbf{real}$
$\textbf{Beta} : (\mathsf{a} : \textbf{real} * \mathsf{b} : \textbf{real}) \to \textbf{real}$
$\textbf{Gamma} : (\mathsf{shape} : \textbf{real} * \mathsf{scale} : \textbf{real}) \to \textbf{real}$

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
V	value
$V_1 \otimes V_2$	arithmetic or logical operator
$V.1$	left projection from pair
$V.2$	right projection from pair
$\textbf{if } V \textbf{ then } M_1 \textbf{ else } M_2$	conditional
$\textbf{let } x = M \textbf{ in } N$	let (scope of x is N)
$\textbf{random } (D(V))$	primitive distribution
$\textbf{observe } V$	observation

In the discrete case, Fun has a standard *sampling semantics*; the formal semantics for the general case comes later. A run of a closed expression M is the process of evaluating M to a value. The evaluation of most expressions is standard, apart from sampling and observation.

To run **random** $(D(V))$, where $V = (c_1, \ldots, c_n)$, choose a value c at random, with probability given by the distribution $D(c_1, \ldots, c_n)$, and return c.

To run **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 sampling semantics of an expression is the conditional probability of returning a particular value, given a valid run.

(Boolean observations are akin to assume statements in assertion-based program specifications, where runs of a program are ignored if an assumed formula is false.)

Example: Two Coins, Not Both Tails

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

The subexpression **random** (**Bernoulli**(0.5)) generates **true** or **false** with equal likelihood. The whole expression has 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 sampling semantics of this expression is a probability distribution assigning probability $1/3$ to the values (**true**, **false**), (**false**, **true**), and (**true**, **true**), but probability 0 to the value (**false**, **false**).

The sampling 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.

Thus, 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?*

2.2 Syntactic Conventions and Monomorphic Typing Rules

We identify phrases of syntax up to consistent renaming of bound variables. Let $\mathrm{fv}(\phi)$ be the set of variables occurring free in phrase ϕ. Let $\phi\,\{{}^{\psi}\!/_{x}\}$ be the outcome of substituting phrase ψ for each free occurrence of variable x in phrase ϕ. 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 $n = 1$, $t = t_1$; and when $n > 1$, $t = t_1 * (t_2 * \cdots * t_n)$. 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. Let $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ if $\Gamma = \varepsilon, x_1 : t_1, \ldots, x_n : t_n$. We sometimes use the notation $\overline{x : t}$ for $\Gamma = \varepsilon, x_1 : t_1, \ldots, x_n : t_n$ where $\bar{x} = x_1, \ldots, x_n$ and $\bar{t} = t_1, \ldots, t_n$.

The typing rules for this monomorphic first-order language are standard.

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

(FUN OPERATOR)
$$\frac{\otimes : b_1, b_2 \to b_3 \quad \Gamma \vdash V_1 : b_1 \quad \Gamma \vdash V_2 : b_2}{\Gamma \vdash V_1 \otimes V_2 : b_3}$$

(FUN RANDOM)
$$\frac{D : (x_1 : b_1 * \cdots * x_n : b_n) \to b \quad \Gamma \vdash V : (b_1 * \cdots * b_n)}{\Gamma \vdash \textbf{random}\ (D(V)) : b}$$

(FUN OBSERVE)
$$\frac{\Gamma \vdash V : b}{\Gamma \vdash \textbf{observe}\ V : \textbf{unit}}$$

3 Semantics as Measure Transformers

If we can only sample from discrete distributions, the semantics of Fun is straightforward. In our technical report, we formalize the sampling semantics of the previous section as a small-step operational semantics for the fragment of Fun where every **random** expression takes the form **random** (**Bernoulli**(c)) for some real $c \in (0,1)$. A reduction $M \to^p M'$ means that M reduces to M' with non-zero probability p.

We cannot give such a semantics to expressions that sample from continuous distributions, such as **random** (**Gaussian**$(1,1)$), since the probability of any particular sample is zero. A further difficulty is the need to observe events with probability zero, a common situation in machine learning. For example, consider the naive Bayesian classifier, a common, simple probabilistic model. 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, assumed to be normally distributed around the class mean. The outcome of the training is the posterior weight distributions for the different classes.

Naive Bayesian Classifier, Single Feature Training:

```
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
```

Above, the call to weight Glass .18 modifies the distribution of the variable Glass. 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 [12]. To avoid this issue, we instead observe the random variable x−y of type **real**, at the value 0.

To give a formal semantics to such observations, as well as to mixtures of continuous and discrete distributions, we turn to measure theory, following standard sources [3]. Two basic concepts are measurable spaces and measures. A measurable space is a set of values equipped with a collection of *measurable* subsets; these measurable sets generalize the events of discrete probability. A finite *measure* is a function that assigns a numeric size to each measurable set; measures generalize probability distributions.

3.1 Types as Measurable Spaces

We let Ω range over sets of possible outcomes; in our semantics Ω will range over $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$, $\mathbb{Z}$, $\mathbb{R}$, and finite Cartesian products of these sets. 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; the elements of $\mathcal{M}$ are called *measurable sets*. 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 the function $f : \Omega \to \Omega'$ is *measurable* if and only if for all $A \in \mathcal{M}'$, $f^{-1}(A) \in \mathcal{M}$, where the *inverse image* $f^{-1} : \mathcal{P}(\Omega') \to \mathcal{P}(\Omega)$ is given by $f^{-1}(A) \triangleq \{\omega \in \Omega \mid f(\omega) \in A\}$. We write $f^{-1}(x)$ for $f^{-1}(\{x\})$ when $x \in \Omega'$.

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

Semantics of Types as Measurable Spaces:

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

The set $\sigma_{\mathbb{R}}(\{[a,b] \mid a,b \in \mathbb{R}\})$ in the definition of $\mathcal{T}[\![\mathbf{real}]\!]$ 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 $f : \mathbf{V}_t \to \mathbf{V}_u$ is measurable, that is, that $f^{-1}(B) \in \mathcal{M}_t$ for all $B \in \mathcal{M}_u$.

3.2 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 $\mu(\cup_i A_i) = \sum_i \mu(A_i)$. We write $|\mu| \triangleq \mu(\Omega)$. Let $\mathtt{M}\,t$ be the set of finite measures on the measurable space $\mathcal{T}[\![t]\!]$. We make use of the following constructions on measures.

- Given a function $f : t \to u$ and a measure $\mu \in \mathtt{M}\,t$, there is a measure $\mu f^{-1} \in \mathtt{M}\,u$ given by $(\mu f^{-1})(B) \triangleq \mu(f^{-1}(B))$.
- 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 $(\mu_1 + \mu_2)(A) \triangleq \mu_1(A) + \mu_2(A)$.
- The (independent) product $(\mu_1 \times \mu_2)$ of two measures is also definable, and satisfies $(\mu_1 \times \mu_2)(A \times B) = \mu_1(A) \cdot \mu_2(B)$. (Existence and uniqueness follows from the Hahn-Kolmogorov theorem.)
- Given a measure μ on the measurable space $\mathcal{T}[\![t]\!]$, a measurable set $A \in \mathcal{M}_t$ and a function $f : t \to \mathbf{real}$, we write $\int_A f\,d\mu$ or equivalently $\int_A f(x)\,d\mu(x)$ for standard (Lebesgue) integration. This integration is always well-defined if μ is finite and f is non-negative and bounded from above.

– Given a measure μ on a measurable space $\mathcal{T}[\![t]\!]$ let a function $\dot{\mu} : t \to$ **real** be a *density* for μ iff $\mu(A) = \int_A \dot{\mu}\, d\lambda$ for all $A \in \mathcal{M}$, where λ is the standard Lebesgue measure on $\mathcal{T}[\![t]\!]$. (We also use λ-notation for functions, but we trust any ambiguity is easily resolved.)

Standard Distributions Given a closed well-typed Fun expression **random** $(D(V))$ of base type b, we define a corresponding finite measure $\mu_{D(V)}$ on measurable space $\mathcal{T}[\![b]\!]$.

In the discrete case, we first define probability masses $D(V)\ c$ of single elements, and hence of singleton sets, and then define the measure $\mu_{D(V)}$ as a countable sum.

Masses $D(V)\ c$ and Measures $\mu_{D(V)}$ for Discrete Probability Distributions:

Bernoulli(p) **true** $\triangleq p$	if $0 \le p \le 1$, 0 otherwise
Bernoulli(p) **false** $\triangleq 1 - p$	if $0 \le p \le 1$, 0 otherwise
Binomial$(n, p)\ i \triangleq \binom{i}{n} p^i / n!$	if $0 \le p \le 1$, 0 otherwise
DiscreteUniform$(m)\ i \triangleq 1/m$	if $0 \le i < m$, 0 otherwise
Poisson$(l)\ n \triangleq e^{-l} l^n / n!$	if $l, n \ge 0$, 0 otherwise
$\mu_{D(V)}(A) \triangleq \sum_i D(V)\ c_i$	if $A = \bigcup_i \{c_i\}$ for pairwise distinct c_i

In the continuous case, we first define probability densities $D(V)\ r$ at individual elements r. and then define the measure $\mu_{D(V)}$ as an integral. Below, we write **G** for the standard Gamma function, which on naturals n satisfies $\mathbf{G}(n) = (n-1)!$.

Densities $D(V)\ r$ and Measures $\mu_{D(V)}$ for Continuous Probability Distributions:

Gaussian$(m, v)\ r \triangleq e^{-(r-m)^2/2v} / \sqrt{2\pi v}$	if $v > 0$, 0 otherwise
Gamma$(s, p)\ r \triangleq r^{s-1} e^{-pr} p^s / \mathbf{G}(s)$	if $r, s, p > 0$, 0 otherwise
Beta$(a, b)\ r \triangleq r^{a-1}(1-r)^{b-1} \mathbf{G}(a+b) / (\mathbf{G}(a)\mathbf{G}(b))$	
	if $a, b \ge 0$ and $0 \le r \le 1$, 0 otherwise
$\mu_{D(V)}(A) \triangleq \int_A D(V)\, d\lambda$	where λ is the Lebesgue measure

The Dirac δ measure is defined on the measurable space $\mathcal{T}[\![b]\!]$ for each base type b, and is given by $\delta_c(A) \triangleq 1$ if $c \in A$, 0 otherwise. We write δ for $\delta_{0.0}$.

The notion of density can be generalized as follows, yielding an unnormalized counterpart to conditional probability. Given a measure μ on $\mathcal{T}[\![t]\!]$ and a measurable function $p : t \to b$, we consider the family of events $p(x) = c$ where c ranges over $\mathbf{V}_b$. We define $\dot{\mu}[A||p = c] \in \mathbb{R}$ (the μ-density at $p = c$ of A) following [8], by:

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

$\dot{\mu}[A||p = c] \triangleq \lim_{i\to\infty} \mu(A \cap p^{-1}(B_i)) / \int_{B_i} 1\, d\lambda$ if the limit exists
and is the same for all sequences $\{B_i\}$ of closed sets converging regularly to c.

Where defined, letting $A \in \mathcal{M}_a, B \in \mathcal{M}_b$, conditional density satisfies the equation

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

In particular, we have $\dot{\mu}[A||p=c]=0$ if b is discrete and $\mu(p^{-1}(c))=0$. To show that our definition of conditional density generalizes the notion of density given above, we have that if μ has a continuous density $\dot{\mu}$ on some neighbourhood of $p^{-1}(c)$ then

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

3.3 Measure Transformers

We will now recast some standard theorems of measure theory as a library of combinators, that we will later use to give semantics to probabilistic languages. A *measure transformer* is a function from finite measures to finite measures. We let $t \rightsquigarrow u$ be the set of functions $\mathtt{M}\,t \to \mathtt{M}\,u$. We use the following combinators on measure transformers in the formal semantics of our languages.

Measure Transformer Combinators:

$\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$
$\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$
$\mathtt{choose} \in (\mathbf{V}_t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$
$\mathtt{extend} \in (\mathbf{V}_t \to \mathtt{M}\,u) \to (t \rightsquigarrow (t * u))$
$\mathtt{observe} \in (t \to b) \to (t \rightsquigarrow t)$

The definitions of these combinators occupy the remainder of this section. We recall that μ denotes a measure and A a measurable set, of appropriate types.

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

To sequentially compose two measure transformers we use standard function composition, defining $\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$ as $T \ggg U \triangleq U \circ T$.

The combinator $\mathtt{choose} \in (\mathbf{V}_t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$ makes a conditional choice between measure transformers, if its first argument is measurable and has finite range. Intuitively, $\mathtt{choose}\ K\ \mu$ first splits $\mathbf{V}_t$ into the equivalence classes modulo K. For each equivalence class, we then run the corresponding measure transformer on μ restricted to the class. Finally, the resulting finite measures are added together, yielding a finite measure. We let $\mathtt{choose}\ K\ \mu\ A \triangleq \sum_{T\in\text{range}(K)} T(\mu|_{K^{-1}(T)})(A)$. In particular, if K is a binary choice mapping all elements of B to T_B and all elements of $C=\overline{B}$ to T_C, we have $\mathtt{choose}\ K\ \mu\ A = T_B(\mu|_B)(A)+T_C(\mu|_C)(A)$. (In fact, our only uses of $\mathtt{choose}$ in this paper are in the semantics of conditional expressions in Fun and conditional statements in Imp, and in each case the argument K to $\mathtt{choose}$ is a binary choice.)

The combinator $\mathtt{extend} \in (\mathbf{V}_t \to \mathtt{M}\,u) \to (t \rightsquigarrow (t * u))$ extends the domain of a measure using a function yielding measures. It is reminiscent of creating a dependent pair, since the distribution of the second component depends on the value of the first. For $\mathtt{extend}\,m$ to be defined, we require that for every $A \in \mathcal{M}_u$, the function $f_A \triangleq \lambda x.m(x)(A)$ is measurable, non-negative and bounded from above. This will always be the case in our semantics for Fun, since we only use the standard distributions for m above. We let $\mathtt{extend}\,m\ \mu\ AB \triangleq \int_{\mathbf{V}_t} m(x)(\{y \mid (x,y) \in AB\})\,d\mu(x)$, where we integrate over the first component (call it x) with respect to the measure μ, and the integrand is the measure $m(x)$ of the set $\{y \mid (x,y) \in A\}$ for each x.

The combinator $\texttt{observe} \in (t \to b) \to (t \rightsquigarrow t)$ conditions a measure over $\mathcal{T}[\![t]\!]$ on the event that an indicator function of type $t \to b$ is zero. Here observation is *unnormalized* conditioning of a measure on an event. We define:

$$\texttt{observe}\ p\ \mu\ A \triangleq \begin{cases} \dot{\mu}[A||p=0_b] & \text{if } \mu(p^{-1}(0_b)) = 0 \\ \mu(A \cap p^{-1}(0_b)) & \text{otherwise} \end{cases}$$

As an example, if $p : t \to \mathbf{bool}$ is a predicate on values of type t, we have

$$\texttt{observe}\ p\ \mu\ A = \mu(A \cap \{x \mid p(x) = \mathbf{true}\}).$$

In the continuous case, if $\mathbf{V}_t = \mathbb{R} \times \mathbb{R}^k$, $p = \lambda(y,x).(y-c)$ and μ has density $\dot{\mu}$ then

$$\texttt{observe}\ p\ \mu\ A = \int_A \mu(y,x)\, d(\delta_c \times \lambda)(y,x) = \int_{\{x|(c,x)\in A\}} \dot{\mu}(c,x)\, d\lambda(x).$$

Notice that $\texttt{observe}\ p\ \mu\ A$ can be greater than $\mu(A)$, for which reason we cannot restrict ourselves to transformation of (sub-)probability measures.

3.4 Measure 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 may draw a parallel to while programs. In this setting, we can give a denotation to a program as a function from variable valuations to a return value and a variable valuation. Similarly, we give semantics to an open Fun term by mapping a measure over assignments to the term's free variables to a joint measure of the term's return value and assignments to its free variables. This choice is a generalization of the (discrete) semantics of pWHILE [2].

First, we define a data structure for an evaluation environment assigning values to variable names, and corresponding operations. Given an environment $\Gamma = x_1{:}t_1, \ldots, x_n{:}t_n$, we let $\texttt{S}\langle\Gamma\rangle$ be the set of states, or finite maps $s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$ such that for all $i = 1, \ldots, n$, $\varepsilon \vdash V_i : t_i$. We let $\mathcal{T}[\![\texttt{S}\langle\Gamma\rangle]\!] \triangleq \mathcal{T}[\![t_1 * \cdots * t_n]\!]$ be the measurable space of states in $\texttt{S}\langle\Gamma\rangle$. We define $\mathrm{dom}(s) \triangleq \{x_1, \ldots, x_n\}$. We define the following operators.

Auxiliary Operations on States and Pairs:

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

We apply these combinators to give a semantics to Fun programs as measure 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, and $\mathcal{A}[\![M]\!]$ gives the measure transformer denoted by M.

Measure Transformer Semantics of Fun:

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

$$
\begin{aligned}
&\mathcal{A}[\![V]\!] \triangleq \texttt{pure}\ \lambda s.(s, \mathcal{V}[\![V]\!]\ s)\\
&\mathcal{A}[\![V_1 \otimes V_2]\!] \triangleq \texttt{pure}\ \lambda s.(s, ((\mathcal{V}[\![V_1]\!]\ s) \otimes (\mathcal{V}[\![V_2]\!]\ s)))\\
&\mathcal{A}[\![V.1]\!] \triangleq \texttt{pure}\ \lambda s.(s, \texttt{fst}(\mathcal{V}[\![V]\!]\ s))\\
&\mathcal{A}[\![V.2]\!] \triangleq \texttt{pure}\ \lambda s.(s, \texttt{snd}(\mathcal{V}[\![V]\!]\ s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N]\!] \triangleq \texttt{choose}\ \lambda s.\texttt{if}\ \mathcal{V}[\![V]\!]\ s\ \texttt{then}\ \mathcal{A}[\![M]\!]\ \texttt{else}\ \mathcal{A}[\![N]\!]\\
&\mathcal{A}[\![\textbf{random}\ (D(V))]\!] \triangleq \texttt{extend}\ \lambda s.\mu_{D(\mathcal{V}[\![V]\!]\ s)}\\
&\mathcal{A}[\![\textbf{observe}\ V]\!] \triangleq (\texttt{observe}\ \lambda s.\mathcal{V}[\![V]\!]\ s) \ggg \texttt{pure}\ \lambda s.(s, \texttt{()})\\
&\mathcal{A}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] \triangleq \mathcal{A}[\![M]\!] \ggg\\
&\qquad\qquad \texttt{pure}\ (\texttt{add}\ x) \ggg \mathcal{A}[\![N]\!] \ggg \texttt{pure}\ \lambda(s,y).((\texttt{drop}\ \{x\}\ s), y)
\end{aligned}
$$

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 measure 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`. A primitive distribution **random** $(D(V))$ extends the state measure with a value drawn from the distribution D, with parameters V depending on the current state. An observation **observe** V modifies the current measure by restricting it to states where V is zero. It is implemented with the `observe` combinator, and it always returns the unit value. 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`.

Lemma 1. *If* $s : \mathtt{S}\langle\Gamma\rangle$ *and* $\Gamma \vdash V : t$ *then* $\mathcal{V}[\![V]\!]\ s \in \mathbf{V}_t$.

Lemma 2. *If* $\Gamma \vdash M : t$ *then* $\mathcal{A}[\![M]\!] \in \mathtt{S}\langle\Gamma\rangle \rightsquigarrow (\mathtt{S}\langle\Gamma\rangle * t)$.

The measure transformer semantics of Fun is hard to use directly, except in the case of discrete measures where they can be directly implemented: a naive implementation of $\mathtt{M}\langle\mathtt{S}\langle\Gamma\rangle\rangle$ is as a map assigning a probability to each possible variable valuation. If there are N variables, each sampled from a Bernoulli distribution, in the worst case there are 2^N paths to be explored in the computation, each of which corresponds to a variable valuation. In this simple case, the measure transformer semantics of closed programs also coincides with the sampling semantics. We write $\mathrm{P}_M[\mathsf{value} = V \mid \mathsf{valid}]$ for the probability that a run of M returns V given that all observations in the run succeed.

Theorem 1. *Suppose* $\varepsilon \vdash M : t$ *for some* M *only using* **Bernoulli** *distributions. If* $\mu = \mathcal{A}[\![M]\!]\ \delta_{()}$ *and* $\varepsilon \vdash V : t$ *then* $\mathrm{P}_M[\mathsf{value} = V \mid \mathsf{valid}] = \mu(\{V\})/|\mu|$.

A consequence of the theorem is that our measure transformer semantics is a generalization of the sampling semantics for discrete probabilities. For this theorem to hold, it is critical that `observe` denotes unnormalized conditioning (filtering). Otherwise programs that perform observations inside the branches of conditional expressions would have undesired semantics. As the following example shows, the two program fragments **observe** (x=y) and **if** x **then observe** (y=**true**) **else observe** (y=**false**) would have different measure transformer semantics although they have the same sampling semantics.

Simple Conditional Expression: M_{if}

```
let x = sample (Bernoulli(0.5))
let y = sample (Bernoulli(0.1))
if x then observe (y=true) else observe (y=false)
y
```

In the sampling semantics, the two valid runs are when x and y are both **true** (with probability 0.05), and both **false** (with probability 0.45), so we have $\mathrm{P}[\textbf{true} \mid \mathsf{valid}] = 0.1$ and $\mathrm{P}[\textbf{false} \mid \mathsf{valid}] = 0.9$.

If, instead of the unnormalized definition $\texttt{observe}\ p\ \mu\ A = \mu(A \cap \{x \mid p(x)\})$, we had either of the flawed definitions

$$\texttt{observe}\ p\ \mu\ A = \frac{\mu(A \cap \{x \mid p(x)\})}{\mu(\{x \mid p(x)\})} \text{ or } |\mu|\frac{\mu(A \cap \{x \mid p(x)\})}{\mu(\{x \mid p(x)\})}$$

then $\mathcal{A}[\![M_{\text{if}}]\!]\ \delta_{()}\ \{\textbf{true}\} = \mathcal{A}[\![M_{\text{if}}]\!]\ \delta_{()}\ \{\textbf{false}\}$, which would invalidate the theorem.

Let $M' = M_{\text{if}}$ with **observe** $(x = y)$ substituted for the conditional expression. With the actual or either of the flawed definitions of `observe` we have $\mathcal{A}[\![M']\!]\ \delta_{()}\ \{\textbf{true}\} = (\mathcal{A}[\![M']\!]\ \delta_{()}\ \{\textbf{false}\})/9$.

4 Semantics as Factor Graphs

A naive implementation of the measure transformer semantics of the previous section would work directly with measures of states, whose size 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* [18], whose size will be linear in the size of the program. We define this semantics in two steps. We first compile the Fun program into a program in the simple imperative language Imp, and then the Imp program itself has a straightforward semantics as a factor graph. Our semantics formalizes the way in which our implementation maps F# programs to Csoft programs, which are evaluated by Infer.NET by constructing suitable factor graphs. The implementation advantage of translating F# to Csoft, over simply generating factor graphs directly [22], is that the translation preserves the structure of the input model (including array processing in our full language), which can be exploited by the various inference algorithms supported by Infer.NET.

4.1 Imp: An Imperative Core Calculus

Imp is an imperative language, based on the static single assignment (SSA) intermediate form. It is a sublanguage of Csoft, the input language of Infer.NET [25], and is intended to have a simple semantics as a factor graph. A composite statement C is a sequence of statements, each of which either stores the result of a primitive operation in a location, observes the contents of 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:

$l, l', \ldots$	Locations (variables) in global store
$E, F ::= c \mid l \mid (l \otimes l)$	Expression
$I ::=$	Statement
$\quad l \leftarrow E$	assignment
$\quad l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)$	random assignment
$\quad \mathbf{observe}_b\ l$	observation
$\quad \mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2$	conditional
$C ::= \mathbf{nil} \mid I \mid (C;C)$	Composite Statement

When making an observation $\mathbf{observe}_b$, we make explicit the type b of the observed location. In the form $\mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2$, the environments Σ_1 and Σ_2 declare the local variables assigned by the **then** branch and the **else** branch, respectively. These annotations simplify type checking and denotational semantics.

The typing rules for Imp are standard. We consider Imp typing environments Σ to be a special case of Fun environments Γ, 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 environment Σ, yielding additional bindings Σ'.

Part of the Type System for Imp: $\Sigma \vdash C : \Sigma'$

(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)
$$\frac{\Sigma \vdash \diamond}{\Sigma \vdash \mathbf{nil} : \varepsilon}$$

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

(IMP OBSERVE)
$$\frac{\Sigma \vdash l : b}{\Sigma \vdash \mathbf{observe}_b\ l : \varepsilon}$$

(IMP IF)
$$\frac{\Sigma \vdash l : \mathbf{bool} \quad \Sigma \vdash C_1 : \Sigma'_1 \quad \Sigma \vdash C_2 : \Sigma'_2 \quad \{\Sigma'_i\} = \{\Sigma_i, \Sigma'\}}{\Sigma \vdash \mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2 : \Sigma'}$$

4.2 Translating from Fun to Imp

The translation from Fun to Imp is a mostly routine compilation of functional code to imperative code. The main point of interest is that Imp locations only hold values of base type, while Fun variables may hold tuples. We rely on *patterns* p and *layouts* ρ to track the Imp locations corresponding to Fun environments. The technical report has the detailed definition of the following notations.

Notations for the Translation from Fun to Imp:

$p ::= l \mid () \mid (p, p)$	pattern: group of Imp locations to represent Fun value
$\rho ::= (x_i \mapsto p_i)^{i \in 1..n}$	layout: finite map from Fun variables to patterns
$\Sigma \vdash p : t$	in environment Σ, pattern p represents Fun value of type t
$\Sigma \vdash \rho : \Gamma$	in environment Σ, layout ρ represents environment Γ
$\rho \vdash M \Rightarrow C, p$	given ρ, expression M translates to C and pattern p

4.3 Factor Graphs

A factor graph [18] 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 [27, 38] as well as sampling techniques [15, Chapter 12]. We use factor graphs with *gates* [26] for modelling if-then-else clauses; gates introduce second-order edges in the graph.

Factor Graphs:

$G ::= \mathsf{new}\ \overline{x:b}\ \mathsf{in}\ \{e_1,\dots,e_m\}$	Graph
$x,y,z,\dots$	Nodes (random variables)
$e ::=$	Edge
$\quad \mathsf{Equal}(x,y)$	$\quad$ equality ($x=y$)
$\quad \mathsf{Constant}_c(x)$	$\quad$ constant ($x=c$)
$\quad \mathsf{Binop}_{\otimes}(x,y,z)$	$\quad$ binary operator ($x=y\otimes z$)
$\quad \mathsf{Sample}_D(x,y_1,\dots,y_n)$	$\quad$ sampling ($x\sim D(y_1,\dots,y_n)$)
$\quad \mathsf{Gate}(x,G_1,G_2)$	$\quad$ gate (if x then G_1 else G_2)

In a graph $\mathsf{new}\ \overline{x:b}\ \mathsf{in}\ \{e_1,\dots,e_m\}$, the variables x_i are bound; graphs are identified up to consistent renaming of bound variables. We write $\{e_1,\dots,e_m\}$ for $\mathsf{new}\ \varepsilon\ \mathsf{in}\ \{e_1,\dots,e_m\}$. We write $\mathrm{fv}(G)$ for the variables occurring free in G. Here is an example factor graph G_{E}. (The corresponding Fun source code is listed in the technical report.)

Factor Graph for Epidemiology Example:

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

A factor graph typically 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 slightly more general semantics of factor graphs as measure transformers; the input measure 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.

Semantics of Factor Graphs: $\mathcal{J}[\![G]\!]_{\Sigma}^{\Sigma'} \in \mathtt{S}\langle\Sigma\rangle \rightsquigarrow \mathtt{S}\langle\Sigma,\Sigma'\rangle$

$$
\begin{aligned}
&\mathcal{J}[\![G]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \triangleq \textstyle\int_A (\mathcal{J}[\![G]\!]\ \ s)\ d(\mu\times\lambda)(s)\\
&\mathcal{J}[\![\mathsf{new}\ \overline{x:b}\ \mathsf{in}\ \{\overline{e}\}]\!]\ s \triangleq \textstyle\int_{\mathbf{V}_{*_i b_i}} \prod_j (\mathcal{J}[\![e_j]\!]\ (s,\overline{x}))\, d\lambda(\overline{x})\\
&\mathcal{J}[\![\mathsf{Equal}(l,l')]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ l'\ s)\\
&\mathcal{J}[\![\mathsf{Constant}_c(l)]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = c)\\
&\mathcal{J}[\![\mathsf{Binop}_{\otimes}(l,w_1,w_2)]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ w_1\ s \otimes \mathtt{lookup}\ w_2\ s)
\end{aligned}
$$

$$\mathcal{J}[\![\mathsf{Sample}_D(l,v_1,\dots,v_n)]\!]\ s \triangleq \mu_{D(\mathtt{lookup}\ v_1\ s,\dots,\mathtt{lookup}\ v_n\ s)}(\mathtt{lookup}\ l\ s)$$
$$\mathcal{J}[\![\mathsf{Gate}(v,G_1,G_2)]\!]\ s \triangleq (\mathcal{J}[\![G_1]\!]\ s)^{[\mathtt{lookup}\ v\ s]}(\mathcal{J}[\![G_2]\!]\ s)^{[\neg\mathtt{lookup}\ v\ s]}$$

4.4 Factor Graph Semantics for Imp

An Imp statement has a straightforward semantics as a factor graph. Here, observation is defined by the value of the variable being the constant 0_b.

Factor Graph Semantics of Imp: $G = \mathcal{G}[\![C]\!]$

$$\mathcal{G}[\![\mathbf{nil}]\!] \triangleq \varnothing$$
$$\mathcal{G}[\![C_1;C_2]\!] \triangleq \mathcal{G}[\![C_1]\!] \cup \mathcal{G}[\![C_2]\!]$$
$$\mathcal{G}[\![l \leftarrow c]\!] \triangleq \{\mathsf{Constant}_c(l)\}$$
$$\mathcal{G}[\![l \leftarrow l']\!] \triangleq \{\mathsf{Equal}(l,l')\}$$
$$\mathcal{G}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \{\mathsf{Binop}_\otimes(l,l_1,l_2)\}$$
$$\mathcal{G}[\![l \overset{s}{\leftarrow} D(l_1,\dots,l_n)]\!] \triangleq \{\mathsf{Sample}_D(l,l_1,\dots,l_n)\}$$
$$\mathcal{G}[\![\mathbf{observe}_b\ l]\!] \triangleq \{\mathsf{Constant}_{0_b}(l)\}$$
$$\mathcal{G}[\![\mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2]\!] \triangleq \{\mathsf{Gate}(l, \mathsf{new}\ \Sigma_1\ \mathsf{in}\ \mathcal{G}[\![C_1]\!], \mathsf{new}\ \Sigma_2\ \mathsf{in}\ \mathcal{G}[\![C_2]\!])\}$$

The following theorem asserts that the semantics of Fun coincides with the semantics of Imp for compatible measures, which are defined as follows. If $T : t \rightsquigarrow u$ is a measure transformer composed from the combinators of Section 3 and $\mu \in \mathtt{M}\,t$, 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 either that f is discrete or that μ has a continuous density on some ε-neighbourhood of $f^{-1}(0.0)$.

The statement of the theorem needs some additional notation. If $\Sigma \vdash p : t$ and $s \in \mathtt{S}\langle\Sigma\rangle$, we write $p\ s$ for the reconstruction of an element of $\mathcal{T}[\![t]\!]$ by looking up the locations of p in the state s. We define as follows operations `lift` and `restrict` to translate between states consisting of Fun variables ($\mathtt{S}\langle\Gamma\rangle$) and states consisting of Imp locations ($\mathtt{S}\langle\Sigma\rangle$), where flatten takes a mapping from patterns to values to a mapping from locations to base values.

$$\mathtt{lift}\ \rho \triangleq \lambda s.\ \mathrm{flatten}\,\{\rho(x) \mapsto \mathcal{V}[\![x]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$$
$$\mathtt{restrict}\ \rho \triangleq \lambda s.\ \{x \mapsto \mathcal{V}[\![\rho(x)]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$$

Theorem 2. *If $\Gamma \vdash M : t$ and $\Sigma \vdash \rho : \Gamma$ and $\rho \vdash M \Rightarrow C, p$ and measure $\mu \in \mathtt{M}\langle\mathtt{S}\langle\Gamma\rangle\rangle$ is compatible with $\mathcal{A}[\![M]\!]$ then there exists Σ' such that $\Sigma \vdash C : \Sigma'$ and:*
$\mathcal{A}[\![M]\!]\ \mu = (\mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]_{\Sigma}^{\Sigma'} \ggg \mathtt{pure}\ (\lambda s.\ (\mathtt{restrict}\ \rho\ s, p\ s)))\ \mu.$

Proof. Via a direct measure transformer semantics for Imp. The proof is by induction on the typing judgments $\Gamma \vdash M : t$ and $\Sigma \vdash C : \Sigma'$. □

5 Implementation Experience

We implemented a compiler from Fun to Imp in F#. We wrote two backends for Imp: an exact inference algorithm based on a direct implementation of measure transformers for discrete measures, and an approximating inference algorithm for continuous measures, using Infer.NET [25]. Translating Imp to Infer.NET is relatively straightforward, and amounts to a syntax-directed series of calls to Infer.NET's object-oriented API.

We have statistics on a few of the examples we have 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 is a tournament ranking model, and adPredictor is a simplified version of a model to predict the likelihood that a display advertisment will be clicked. In 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%) in Infer.NET, performing inference. AdPredictor spends most of the time in pre-processing (58%), and only 40% 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. On the one hand, Infer.NET supports a limited set of operations on specific combinations of probabilistic and deterministic arguments. Our discrete backend, on the other hand, is limited to small models using only finite measures.

6 Related Work

To the best of our knowledge, this paper introduces the first rigorous measure-theoretic semantics shown to be in agreement with a factor graph semantics for a probabilistic language with observation and sampling from continuous distributions. Hence, we lay a firm foundation for inference on probabilistic programs via modern message-passing algorithms on factor graphs.

Formal Semantics of Probabilistic Languages There is a long history of formal semantics for probabilistic languages with sampling primitives, often combined with recursive computation. One of the first semantics is for Probabilistic LCF [35], which augments the core functional language LCF with weighted binary choice, for discrete distributions. Kozen [17] develops a probabilistic semantics for while-programs augmented with random assignment. He develops two provably equivalent semantics; one more operational, and the other a denotational semantics using partially ordered Banach spaces.

Imp is simpler than Kozen's language, as Imp has no unbounded while-statements, so the semantics of Imp need not deal with non-termination. On the other hand, observations are not present in Kozen's language.

Jones and Plotkin [13] investigate the probability monad, and apply it to languages with discrete probabilistic choice. Ramsey and Pfeffer [33] give a stochastic λ-calculus with a measure-theoretic semantics in the probability monad, and provide an embedding within Haskell; they do not consider observations. We can generalize the semantics of **observe** to this setting as filtering in the probability monad (yielding what we may call a sub-probability monad), as long as the events that are being observed are discrete or have non-zero probability. However, zero-probability observations of real variables do not translate easily to the probability monad, as the following example shows. Let N be an expression returning a continuous distribution, for example, **sample** (**Gaussian**(0.0,1.0)), and let $\mathsf{f}\,\mathsf{x}$ = **observe** x. The probability monad semantics of the program **let** $\mathsf{x} = N$ **in** $\mathsf{f}\,\mathsf{x}$ of the stochastic λ-calculus is $[\![N]\!] \gg\!= \lambda y.[\![\mathsf{f}\,\mathsf{x}]\!]\{\mathsf{x} \mapsto y\}$, which yields the measure $\mu(A) = \int_{\mathbb{R}}(\mathbf{M}[\![[\![\mathsf{f}\,\mathsf{x}]\!]\{\mathsf{x} \mapsto y\}]\!])(A)\ d\mathbf{M}[N](y)$. Here the probability $(\mathbf{M}[\![[\![\mathsf{f}\,\mathsf{x}]\!]\{x \mapsto y\}]\!])(A)$ is zero except when $y = 0$, where it is some real number. Since the N-measure of $y = 0$ is zero, the whole integral is zero for all A (in particular $\mu(\mathbb{R}) = 0$), whereas the intended semantics is that x is constrained to be zero with probability 1 (so in particular $\mu(\mathbb{R}) = 1$).

The probabilistic concurrent constraint programming language pcc of Gupta, Jagadeesan, and Panangaden [10] is also intended for describing probability distributions using independent sampling and constraints. Our use of observations corresponds to constraints on random variables in pcc. In the finite case, pcc also relies on a sampling semantics with observation (constraints) denoting filtering. To admit continuous distributions, pcc adds general fixpoints and defines the semantics of a program as the limit of finite unrollings of its fixpoints, if defined. This can lead to surprising results, for example, that the distribution resulting from observing that two uniform distributions are equal may not itself be uniform. In contrast, our goal is an efficient implementation via factor graphs, which led us to work directly with standard distributions and to have a semantics of observation that is independent of the program text.

McIver and Morgan [23] develop a theory of abstraction and refinement for probabilistic while programs, based on weakest preconditions. They reject a subdistribution transformer semantics in order to admit demonic nondeterminism in the language.

We conjecture that Fun and Imp could in principle be conferred semantics within a probabilistic language supporting general recursion, by encoding observations by placing the whole program within a conditional sampling loop, and by encoding Gaussian and other continuous distributions as repeated sampling using recursive functions. Still, our choices in formulating the semantics of Fun and Imp were to include some distributions as primitive, and to exclude recursion; compared to encodings within probabilistic languages with recursion, these choices have these advantages: (1) our measure transformer semantics relies on relatively elementary measure theory, with no need to express non-termination or to compute limits when defining the model; (2) our semantics is compositional rather than relying on a global sampling loop; and (3) our semantics has a direct implementation via message-passing algorithms on factor graphs, with efficient implementations of primitive distributions.

Probabilistic Languages for Machine Learning Koller et al. [16] pioneered the idea of representing a probability distribution using first-order functional programs 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 distributions $\mathrm{P}[A \wedge B], \mathrm{P}[A \wedge \neg B], \mathrm{P}[\neg A]$ which can be used to compute $\mathrm{P}[B \mid A]$.

Park et al. [30] propose $\lambda_\circ$, the first probabilistic language with formal semantics applied to actual machine learning 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.

Infer.NET [25] is a software library that implements the approximate deterministic algorithms expectation propagation [27] and variational message passing [38], as well as Gibbs sampling, a nondeterministic algorithm. Infer.NET models are written in a probabilistic subset of C#, known as Csoft [37]. Csoft allows **observe** on zero probability events, but its semantics has not previously been formalized and it is currently only implemented as an internal language of Infer.NET. IBAL [32] has observations and uses a factor graph semantics, but only works with discrete datatypes and thus does not need advanced probability theory. Moreover, there seems to be no proof that the factor graph denotation of an IBAL program yields the same distribution as the direct semantics, an important goal of the present work. HANSEI [14] is a programming library for OCaml, based on explicit manipulation of discrete probability distributions as lists, and sampling algorithms based on coroutines. HANSEI uses an explicit `fail` statement, which is equivalent to **observe false** and so cannot be used for conditioning on zero probability events.

FACTORIE [22] is a Scala library for explicitly constructing factor graphs. Although there are many Bayesian modelling languages, Csoft and IBAL are the only previous languages implemented by a compilation to factor graphs. Church [9] 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 methods. Blaise [4] supports the compositional construction of sophisticated probabilistic models, and decouples the choice of inference algorithm from the specification of the distribution. WinBUGS [28] is a popular language for explicitly describing distributions suitable for Monte Carlo analysis.

Other Uses of Probabilistic Languages Probabilistic languages with formal semantics find application in many areas apart from machine learning, including databases [6], model checking [19], differential privacy [24, 34], information flow [20], and cryptography [1]. A recent monograph on semantics for labelled Markov processes [29] focuses on bisimulation-based equational reasoning. The syntax and semantics of Imp is modelled on an existing probabilistic language [2] without observations.

Erwig and Kollmansberger [7] describe a library for probabilistic functional programming in Haskell. The library is based on the probability monad, and uses a finite representation suitable for small discrete distributions; the library would not suffice to provide a semantics for Fun or Imp with their continuous and hybrid distributions.

7 Conclusion

Our direct contribution is a rigorous semantics for a probabilistic programming language that also has an equivalent factor graph semantics. We have shown that probabilistic functional programs with iteration over arrays, but without the complexities of general recursion, are a concise representation for complex probability distributions arising in machine learning. An 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 verify the transformations performed by machine learning compilers such as Infer.NET compiler for Csoft? Are there type systems for avoiding zero probability exceptions, or to ensure that we only generate factor graphs that can be handled by our back-end?

Acknowledgements We gratefully acknowledge discussions with Ralf Herbrich, Tom Minka, and John Winn. Comments from Nikhil Swamy, Dimitrios Vytiniotis, and the anonymous reviewers were helpful.

References

1. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002.
2. G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.
3. P. Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.
4. K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, MIT, 2008. Available as Technical Report MIT-CSAIL-TR-2008-044.
5. J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. Technical report, Microsoft Research, 2011.
6. N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
7. M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16(1):21–34, 2006.
8. D. A. S. Fraser, P. McDunnough, A. Naderi, and A. Plante. On the definition of probability densities and sufficiency of the likelihood map. *J. Probability and Mathematical Statistics*, 15:301–310, 1995.
9. 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.
10. V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *POPL*, pages 189–202, 1999.
11. R. Herbrich, T. Minka, and T. Graepel. TrueSkill(TM): A Bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, 2007.
12. E. T. Jaynes. *Probability Theory: The Logic of Science*, chapter 15.7 The Borel-Kolmogorov paradox, pages 467–470. CUP, 2003.

13. C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, pages 186–195. IEEE Computer Society, 1989.
14. O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *UAI*, 2009.
15. D. Koller and N. Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.
16. D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.
17. D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
18. 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.
19. M. Z. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. *ENTCS*, 153(2):5–31, 2006.
20. G. Lowe. Quantifying information flow. In *CSFW*, pages 18–31. IEEE Computer Society, 2002.
21. D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. CUP, 2003.
22. 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).
23. A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems*. Monographs in computer science. Springer, 2005.
24. F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference*, pages 19–30. ACM, 2009.
25. T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from `http://research.microsoft.com/infernet`.
26. T. Minka and J. M. Winn. Gates. In *NIPS*, pages 1073–1080. MIT Press, 2008.
27. T. P. Minka. Expectation Propagation for approximate Bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann, 2001.
28. I. Ntzoufras. *Bayesian Modeling Using WinBUGS*. Wiley, 2009.
29. P. Panangaden. *Labelled Markov processes*. Imperial College Press, 2009.
30. S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL*, pages 171–182. ACM, 2005.
31. A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *IJCAI*, pages 733–740. Morgan Kaufmann, 2001.
32. A. Pfeffer. *Statistical Relational Learning*, chapter The design and implementation of IBAL: A General-Purpose Probabilistic Language. MIT Press, 2007.
33. N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.
34. J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, pages 157–168, 2010.
35. N. Saheb-Djahromi. Probabilistic LCF. In *MFCS*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.
36. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
37. 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.
38. J. M. Winn and C. M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.