Distribution Transformer Semantics for Bayesian Machine Learning

Johannes Borgström[1], Andrew D. Gordon[1], Michael Greenberg[2], James Margetson[1], and Jurgen Van Gael[3]

[1] Microsoft Research
[2] University of Pennsylvania
[3] Microsoft FUSE Labs

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 novel combinators for distribution 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 distributions, and observations of zero-probability events. We compile our core language to a small imperative language that in addition to the distribution transformer semantics also has a straightforward semantics via factor graphs, data structures that enable many efficient inference algorithms. We then 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, pioneered by Koller et al. [15], and recently gaining in popularity [29, 28, 8, 4, 13], of writing Bayesian models as probabilistic programs. In particular, we draw inspiration from Csoft [36], an imperative language with an informal probabilistic semantics. Csoft is the native language of Infer.NET [24], a software library for Bayesian reasoning. A compiler turns Csoft programs into factor graphs [17], data structures that support efficient inference algorithms [14]. 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 [10], 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 our example by generating probability distributions for the players' skills, given three played games (observations).

```
// 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* **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, 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.

Factor graphs 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 [30], 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. However, our semantics based on measure theory makes sense of such observations, and inference is 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. (§2)
- Fun has a rigorous probabilistic semantics based on measure-theoretic distribution transformers. (§3)
- Fun has an efficient implementation: our system compiles Fun to Imp, a subset of Csoft, and then relies on Infer.NET. (§4)

Our main contribution is a framework for distribution transformer semantics, which supports discrete distributions, continuous distributions, and mixtures of the two, together with observations of zero probability events.

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. 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. Our semantics is the first for languages with continuous or hybrid distributions, such as Fun or Imp, that are implemented by deterministic inference via factor graphs.

We designed Fun to be a subset of the F# dialect of ML [35], 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.

We end the paper with a description of related work (§5) and some concluding remarks (§6). A draft technical report available at `http://johannes.borgstroem.org/drafts/bayesTR.pdf` includes: detailed proofs; extensions of Fun, Imp, and our factor graph notations with collection and 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:

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

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

$\&\&, ||, =$: **bool**, **bool** $\to$ **bool** $\quad >, =$: **int**, **int** $\to$ **bool**
$+, -, *, \%, /$: **int**, **int** $\to$ **int** $\quad >$: **real**, **real** $\to$ **bool** $\quad +, -, *$: **real**, **real** $\to$ **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: $D : (x_1 : t_1 * \cdots * x_n : t_n) \to t_{n+1}$

Bernoulli : (success : **real**) $\to$ **bool**
Poisson : (rate : **real**) $\to$ **int**
DiscreteUniform : (max : **int**) $\to$ **int**
Gaussian : (mean : **real** $*$ variance : **real**) $\to$ **real**
Gamma : (shape : **real** $*$ scale : **real**) $\to$ **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
if V **then** M_1 **else** M_2	conditional
let $x = M$ **in** N	let (scope of x is N)
sample $(D(V))$	sample a distribution
observe V	observation

We outline an intuitive sampling semantics for this language; the formal 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 **sample** $(D(V))$, 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 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.

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

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

(FUN OPERATOR)
$$\frac{\otimes : t_1, t_2 \to t_3 \quad \Gamma \vdash V_1 : t_1 \quad \Gamma \vdash V_2 : t_2}{\Gamma \vdash V_1 \otimes V_2 : t_3}$$

(FUN SAMPLE)
$$\frac{D : (x_1 : t_1 * \cdots * x_n : t_n) \to t_{n+1} \quad \Gamma \vdash V : (t_1 * \cdots * t_n)}{\Gamma \vdash \textbf{sample}\ (D(V)) : t_{n+1}}$$

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

3 Semantics as Distribution Transformers

If we can only sample from discrete distributions, the semantics of Fun is straightforward. In our technical report, we provide a small-step operational semantics for the fragment of Fun where every **sample** expression takes the form **sample** (**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 **sample** (**Gaussian**$(1,1)$), since the probability of any particular sample is zero. The standard mathematical framework for dealing with discrete and also continuous probabilities is measure theory [3, 33], which we adopt for the semantics of Fun.

A further issue to address is the problem of conditioning distributions by events with probability zero, a common situation in machine learning. For an 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 follow a normal distribution. 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 [11][4]. 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 continuous and discrete distributions, we turn to measure theory, following standard texts [3, 33].

3.1 Types as Measurable Spaces

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

[4] Our compiler does permit the expression **observe** (`x=y`), as sugar for **observe** (`x−y`).

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 $A \in \mathcal{M}'$, $f^{-1}(A) \in \mathcal{M}$.

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 $()$ for $\varnothing$, the unit value.

Semantics of Types as Measurable Spaces:

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

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

3.2 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 $\mu(\cup_i A_i) = \sum_i \mu(A_i)$. We write $|\mu| \triangleq \mu(\Omega)$. 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 the measurable space $\mathcal{T}[\![t]\!]$. 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 $(\mu_1 + \mu_2)(A) \triangleq \mu_1(A) + \mu_2(A)$.

Standard Distributions Here we define the standard distributions that can occur in Fun expressions. In the discrete case it is sufficient to define the probabilities of single elements (and thus of singleton sets; all others follow from countable additivity).

Discrete Probability Distributions:

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

The named finite continuous distributions have a *probability density* with respect to standard (Lebesgue) integration on $\mathbb{R}$. We write $d\lambda$ for integration with respect to the standard Lebesgue measure. The function $\dot{\mu}$ is a density for μ iff $\mu(A) = \int_A \dot{\mu}\, d\lambda$ for all measurable A. Below, we use the standard Γ function, which satisfies $\Gamma(n) = (n-1)!$.

Densities of Continuous Distributions:

Gaussian$(m,v)\ x \triangleq e^{-(x-m)^2/2v}/\sqrt{2\pi v}$	if $v > 0$, 0 otherwise
Gamma$(s,p)\ x \triangleq x^{s-1}e^{-px}p^s/\Gamma(s)$	if $x,s,p > 0$, 0 otherwise

By abuse of notation, when we write $D(V)$ in a context where a measure is expected, it stands for the measure with density $D(V)$ as defined above. 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

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 *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 use the following combinators on distribution transformers in the formal semantics of our languages.

Distribution Transformer Combinators:

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

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

To lift a pure measurable function to a distribution 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 distribution 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 distribution 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 distribution transformer on μ restricted to the class. Finally, the resulting finite measures are added together, yielding a finite measure. We let $\mathtt{choose}\ K\ \mu\ A \triangleq \sum_{T \in \mathrm{range}(K)} T(\mu|_{K^{-1}(T)})(A)$.

The combinator $\mathtt{extend} \in (\mathbf{V}_t \to \mathtt{D}\,u) \to (t \rightsquigarrow (t * u))$ extends the domain of a measure using a function yielding distributions. It is reminiscent of creating a dependent product, where the distribution over the second component depends on the first. We let $\mathtt{extend}\ m\ \mu\ AB \triangleq \int_{\mathbf{V}_t} m(x)(\{y \mid (x,y) \in AB\})d\mu(x)$.

The combinator $\mathtt{observe} \in (t \to b) \to (t \rightsquigarrow t)$ restricts a measure over $\mathcal{T}[\![t]\!]$ to the event that an indicator function of type $t \to b$ is zero. We here consider restriction as *unnormalized* conditioning of a measure on an event. In general, we define conditioning as follows. Given a finite measure μ on $\mathcal{T}[\![t]\!]$ and a function $p : t \to b$, we consider the

family of events $p(x) = c$ indexed by constant c with $\mathrm{ty}(c) = b$. By the Radon-Nikodym theorem there exists (cf. [3, Ex 33.5]) a family of finite measures $\mu[\cdot||p = c]$ on $\mathcal{T}[\![t]\!]$ with the property that for all $B \in \mathcal{M}_b$,

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

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

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 b is discrete, we then require that $\mu[A||p = c] = 0$ for all A. Case (2): Otherwise, we follow [7], and define conditional probability as $\mu[A||p = c] \triangleq \lim_{n\to\infty} \mu(A \cap p^{-1}(B_n))/\mu(p^{-1}(B_n))$ if the limit exists and is the same for all sequences $\{B_i\}$ of closed Lagrange-measurable sets converging regularly to c. In particular, if μ has a continuous density $\dot{\mu}$ on some neighbourhood of C we get that

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

For the remainder of the paper, we assume that all $\mu[A||p = c]$ satisfy (1) and (2) above.

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:

$$\texttt{observe}\ p\ \mu\ A = \int_A \delta(y - c)\dot{\mu}(y,x)\, d(y,x) = \int_{\{x|(c,x)\in A\}} \dot{\mu}(c,x)\, dx$$

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 draw a parallel to programs in ML with references. In this 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 [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 $\mathtt{S}\langle\Gamma\rangle$ be the set of states, or finite maps $s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$ such that for all $i = 1, \ldots, n$, $\varepsilon \vdash V_i : t_i$. 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 $\mathrm{dom}(s) \triangleq \{x_1, \ldots, x_n\}$. We define the following operators.

Operations on States:

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

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, and $\mathcal{A}[\![M]\!]$ gives the distribution transformer denoted by M.

Distribution Transformer Semantics of Fun:

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

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

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

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`. When sampling, **sample**$(D(V))$ extends the state distribution with a value drawn from the distribution 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. 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 distribution transformer semantics of Fun are hard to use directly, except in the case of finite distributions 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 (which is clearly of exponential size in the number of variables in the worst case). In

this simple case, the distribution 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 M evaluates to V given that all observations in the evaluation succeed.

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

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: e.g., the two program fragments **observe** (x=y) and **if** x **then observe** (y=**true**) **else observe** (y=**false**) that have the same sampling semantics would have different distribution transformer 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 $\mathrm{P}\,[\mathbf{true} \mid \mathsf{valid}] = 0.1$ and $\mathrm{P}\,[\mathbf{false} \mid \mathsf{valid}] = 0.9$.

If we had 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 $\mathcal{A}[\![M_{\mathrm{if}}]\!]\ \delta_{()}\ \{\mathbf{true}\} = \mathcal{A}[\![M_{\mathrm{if}}]\!]\ \delta_{()}\ \{\mathbf{false}\}$, invalidating the theorem.

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

4 Semantics as Factor Graphs

A naive implementation of the distribution transformer semantics of the previous section would work directly with distributions 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* [17], whose size will be linear in the size of the program. define this semantics in two steps: First, we compile the Fun program into a simple imperative language Imp, which has a straightforward semantics as factor graphs. The implementation advantage of translating Fun to Imp, over simply generating factor graphs directly from Fun [20], is that the translation preserves the structure of the input model, which can be exploited by inference algorithms. This allows us to choose from 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 (SSA) intermediate form. It is a sublanguage of Csoft, the input language of Infer.NET [24], 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
$\quad l \xleftarrow{s} D(l_1, \ldots, l_n)$	sampling
$\quad \mathbf{observe}_b\ l$	observation
$\quad \mathbf{if}\ l\ \mathbf{then}_{S_1}\ C_1\ \mathbf{else}_{S_2}\ C_2$	conditional
$C ::=$	Composite Statement
$\quad \mathbf{nil}$	empty statement
$\quad I$	single statement
$\quad C; C$	sequencing

When making an observation $\mathbf{observe}_b$, we make explicit the type b of the observed location. In the form $\mathbf{if}\ l\ \mathbf{then}_{S_1}\ C_1\ \mathbf{else}_{S_2}\ C_2$, 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 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 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:

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

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 $\mathtt{S}\langle\Sigma\rangle$ (where Σ is a special case of Γ).

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

$$\mathcal{I}[\![\textbf{nil}]\!] \triangleq \mathtt{pure}\ \mathtt{id}$$
$$\mathcal{I}[\![C_1; C_2]\!] \triangleq \mathcal{I}[\![C_1]\!] \ggg \mathcal{I}[\![C_2]\!]$$
$$\mathcal{I}[\![l \leftarrow c]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, c)$$
$$\mathcal{I}[\![l \leftarrow l']\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, \mathtt{lookup}\ l'\ s)$$
$$\mathcal{I}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, (\mathtt{lookup}\ l_1\ s \otimes \mathtt{lookup}\ l_2\ s))$$
$$\mathcal{I}[\![l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)]\!] \triangleq \mathtt{extend}\ \lambda s.D(\mathtt{lookup}\ l_1\ s, \ldots, \mathtt{lookup}\ l_n\ s) \ggg \mathtt{pure}\ (\mathtt{add}\ l)$$
$$\mathcal{I}[\![\textbf{observe}_b\ l]\!] \triangleq \mathtt{observe}\ \lambda s.\mathtt{lookup}\ l\ s$$
$$\mathcal{I}[\![\textbf{if}\ l\ \textbf{then}_{S_1}\ C_1\ \textbf{else}_{S_2}\ C_2]\!] \triangleq \mathtt{choose}\ \lambda s.\mathtt{if}\ (\mathtt{lookup}\ l\ s)\ \mathtt{then}\ (T_1 \ggg \mathtt{pure}\ (\mathtt{drop}\ S_1))\ \mathtt{else}\ (T_2 \ggg \mathtt{pure}\ (\mathtt{drop}\ S_2))$$

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 mostly a routine 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. The relation $\Sigma \vdash p : t$ means that in heap typing Σ, the pattern p matches the type t.

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)
$$\frac{}{\rho \vdash x \Rightarrow \textbf{nil}, \rho(x)}$$

(TRANS CONST)
$$\frac{c \neq ()}{\rho \vdash c \Rightarrow l \leftarrow c, l}$$

(TRANS UNIT)
$$\frac{}{\rho \vdash () \Rightarrow \textbf{nil}, ()}$$

(TRANS PAIR)
$$\frac{\rho \vdash V_1 \Rightarrow C_1, p_1 \qquad \rho \vdash V_2 \Rightarrow C_2, p_2}{\rho \vdash (V_1, V_2) \Rightarrow C_1; C_2; (p_1, p_2)}$$

(TRANS PROJ1)
$$\frac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.1 \Rightarrow C, p_1}$$

(TRANS PROJ2)
$$\frac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.2 \Rightarrow C, p_2}$$

(TRANS IF)
$$\frac{\rho \vdash V_1 \Rightarrow C_1, l \quad p_2 \sim p \text{ fresh} \quad \rho \vdash M_2 \Rightarrow C_2, p_2 \quad \rho \vdash M_3 \Rightarrow C_3, p_3}{\rho \vdash (\textbf{if } V_1 \textbf{ then } M_2 \textbf{ else } M_3) \Rightarrow (C_1; \textbf{if } l \textbf{ then } C_2; p \leftarrow p_2 \textbf{ else } C_3; p \leftarrow p_3), p}$$

(TRANS OBSERVE)
$$\frac{\rho \vdash V \Rightarrow C, p}{\rho \vdash \textbf{observe } V \Rightarrow C; \textbf{observe}_b\ p, ()}$$

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

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

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, and hence have 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 shared 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.

The rule (TRANS OBSERVE) translates **observe** by running the commands to generate the value for V and then observing the pattern. (Just as for Fun, we restrict observations to locations of base type—so p is a location l.) The rule (TRANS SAMPLE)

translates sampling in much the same way. By $D(p)$, we mean the flattening of p into a list of locations and passing it to the distribution constructor D.

Finally, the rule (TRANS LET) translates **let** statements by running both expressions in sequence. 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 1. *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 $\Sigma \vdash C : \Sigma'$.*

We define operations `lift` and `restrict` to translate between distributions on Fun variables ($\mathtt{S}\langle\Gamma\rangle$) and distributions on Imp locations ($\mathtt{S}\langle\Sigma\rangle$).

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

We write $p\ s$ for the reconstruction of a p-structured value by looking up its locations in s. Given these notations, we state that the compilation of Fun to Imp preserves the distribution transformer semantics, modulo the pattern and superfluous variables.

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

Proof. By induction on the typing of M. □

4.4 Factor Graphs

A factor graph [17] 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 [23, 37] as well as sampling techniques [14, Chapter 12]. We use factor graphs with *gates* [25] for modelling if-then-else clauses; gates introduce second-order edges in the graph.

Factor Graphs:

$x, y, z, \ldots$	Nodes (random variables)
$G ::= \mathsf{new}\ \bar{x}\ \mathsf{in}\ \{e_1, \ldots, e_n\}$	Graph
$e ::=$	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 ($x = y \otimes z$)
$\quad \mathsf{Sample}_D(x, y_1, \ldots, y_n)$	$\quad$ Sampling ($x \sim D(y_1, \ldots, y_n)$)
$\quad \mathsf{Select}_n(x, v, y_1, \ldots, y_n)$	$\quad$ (De)Multiplexing ($x = y_v$)
$\quad \mathsf{Gate}(x, G_1, G_2)$	$\quad$ 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, which are higher-order edges. All variables free in the branches of the if-statement are also accessible in the graph containing the gate. Here is an example factor graph. (The corresponding Fun source code is listed in the technical report.)

Factor Graph for Epidemiology Example:

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

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. As before, $d\lambda$ denotes integration with respect to the standard Lebesgue measure.

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

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

4.5 Factor Graph Semantics for Imp

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

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

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

The following theorem asserts that the two semantics for Imp coincide for compatible measures, which is defined as follows. If $T : t \rightsquigarrow u$ is a distribution transformer composed from the combinators of Section 3 and $\mu \in \mathtt{D}\, t$, we say that T is *compatible* with

μ if every application of `observe` f to some μ' in the evaluation of $T(\mu)$ satisfies the preconditions of either (1) or (2) above (i.e., f is either discrete or μ has a continuous density on some neighbourhood of $f^{-1}(0.0)$).

Theorem 3. *If* $\Sigma \vdash C : \Sigma'$ *and* $\mu \in \mathtt{D}\langle \mathtt{S}\langle \Sigma \rangle \rangle$ *is compatible with* $\mathcal{I}[\![C]\!]$ *then* $\mathcal{I}[\![C]\!]\ \mu = \mathcal{P}[\![\mathcal{G}[\![C]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu$.

Proof. By induction on the typing of C. □

5 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.

Csoft allows **observe** on zero probability events, but its semantics has not previously been formalized. IBAL [30] 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 [13] 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.

There is a long history of formal semantics for probabilistic languages with sampling primitives, often combined with recursive computation; recent monographs review the development of assertion-based reasoning about discrete randomized programs [21] and the semantics for labelled Markov processes [27]. One of the first semantics is for Probabilistic LCF [34], which augments the core functional language LCF with weighted binary choice, for discrete distributions. Jones and Plotkin [12] investigate the probability monad, and apply it to languages with discrete probabilistic choice. Ramsey and Pfeffer [31] define a stochastic lambda-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 have non-zero probability. However, zero-probability observations do not translate easily to the probability monad, as the following example shows. Let N be an expression returning a continuous distribution, e.g., **sample** (**Gaussian**(0.0,1.0)), and let `f x` = **observe** `x`. The probability monad semantics of the program **let** `x` = N **in** `f x` of the stochastic lambda calculus is $[\![N]\!] \gg\!= \lambda y.[\![\mathtt{f\ x}]\!]\{\mathtt{x} \mapsto y\}$, which yields the measure $\mu(A) = \int_{\mathbb{R}} (\mathbf{M}[\![[\![\mathtt{f\ x}]\!]\{\mathtt{x} \mapsto y\}]\!])(A)\, d\mathbf{M}[N](y)$. Here the probability $(\mathbf{M}[\![[\![\mathtt{f\ x}]\!]\{x \mapsto y\}]\!])(A)$ 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 $\mu(\mathbb{R}) = 1$).

Gupta, Jagadeesan, and Panangaden [9] give a semantics for probabilistic concurrent constraint programming, including recursion; our use of observations correspond to constraints on random variables in their setting. 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. We leave this conjecture for future study. 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 distribution 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.

Koller et al. [15] 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 $\mathrm{P}\left[A \wedge B\right], \mathrm{P}\left[A \wedge \neg B\right], \mathrm{P}\left[\neg A\right]$ which can be used to compute $\mathrm{P}\left[B \mid A\right]$.

Park et al. [28] 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.

Erwig and Kollmansberger [6] describe a library for probabilistic functional programming in Haskell. The library is based on the probability monad, and uses a finite representation suitable for discrete distributions; the library would not suffice to provide a semantics for Fun or Imp with their continuous and mixture distributions.

FACTORIE [20] 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 [8] 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 [4] supports the compositional construction of sophisticated probabilistic models, and decouples the choice of inference algorithm from the specification of the distribution. WinBUGS [26] is a popular langauge for explicitly describing distributions suitable for Monte Carlo analysis.

Probabilistic languages with formal semantics find application in many areas apart from machine learning, including databases [5], model checking [18], differential privacy [22, 32], information flow [19], and cryptography [1]. The syntax and semantics of Imp is modelled on an existing probabilistic language [2] without observations.

6 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#) 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 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. Nikhil Swamy and Dimitrios Vytiniotis commented helpfully.

References

1. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002.
2. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.
3. Patrick Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.
4. Keith A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, MIT, 2008. Available as Technical Report MIT-CSAIL-TR-2008-044.
5. Nilesh N. Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
6. Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16(1):21–34, 2006.
7. D. A. S. Fraser and Amir Naderi. *Research developments in probability and statistics: Festschrift in honor of Madan L. Puri on the Occasion of His 65th Birthday*, chapter On the Definition of Conditional Probability. Brill Academic Publishers, 1996.
8. Noah Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229. AUAI Press, 2008.
9. Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Stochastic processes as concurrent constraint programs. In *POPL*, pages 189–202, 1999.
10. R. Herbrich, T. Minka, and T. Graepel. Trueskill(tm): A Bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, 2007.
11. Edwin Thompson Jaynes. *Probability Theory: The Logic of Science*, chapter 15.7 The Borel-Kolmogorov paradox, pages 467–470. CUP, 2003.
12. C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, pages 186–195. IEEE Computer Society, 1989.
13. Oleg Kiselyov and Chung-chieh Shan. Monolingual probabilistic programming using generalized coroutines. In *UAI*, 2009.

14. Daphne Koller and Nir Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.
15. Daphne Koller, David A. McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.
16. Dexter Kozen. A probabilistic pdl. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.
17. Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.
18. Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Quantitative analysis with the probabilistic model checker PRISM. *ENTCS*, 153(2):5–31, 2006.
19. Gavin Lowe. Quantifying information flow. In *CSFW*, pages 18–31. IEEE Computer Society, 2002.
20. Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs, 2009. Poster at 23rd Annual Conference on Neural Information Processing Systems (NIPS).
21. Annabelle McIver and Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Monographs in computer science. Springer, 2005.
22. Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference*, pages 19–30. ACM, 2009.
23. Thomas P. Minka. Expectation Propagation for approximate Bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann, 2001.
24. Tom Minka, John Winn, John Guiver, and Anitha Kannan. Infer.net 2.3, November 2009. Software available from `http://research.microsoft.com/infernet`.
25. Tom Minka and John M. Winn. Gates. In *NIPS*, pages 1073–1080. MIT Press, 2008.
26. Ioannis Ntzoufras. *Bayesian Modeling Using WinBUGS*. Wiley, 2009.
27. Prakash Panangaden. *Labelled Markov processes*. Imperial College Press, 2009.
28. Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *POPL*, pages 171–182. ACM, 2005.
29. Avi Pfeffer. Ibal: A probabilistic rational programming language. In Bernhard Nebel, editor, *IJCAI*, pages 733–740. Morgan Kaufmann, 2001.
30. Avi Pfeffer. *Statistical Relational Learning*, chapter The design and implementation of IBAL: A General-Purpose Probabilistic Language. MIT Press, 2007.
31. Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.
32. Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, pages 157–168, 2010.
33. Jeffrey S. Rosenthal. *A First Look at Rigorous Probability Theory*. World Scientific, 2nd edition, 2006.
34. N. Saheb-Djahromi. Probabilistic LCF. In *MFCS*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.
35. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
36. John Winn and Tom Minka. Probabilistic programming with Infer.NET. Machine Learning Summer School lecture notes, available at `http://research.microsoft.com/~minka/papers/mlss2009/`, 2009.
37. John M. Winn and Christopher M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.