The Random Conditional Distribution

For Uncertain Distributional Properties

Zenna Tavares
BCS, CSAIL
MIT

Xin Zhang
CSAIL
MIT

Edgar Minaysan
Princeton

Javier Burroni
UMass Amherst

Rajesh Ranganath
Courant Institute
NYU

Armando Solar Lezama
CSAIL
MIT

Abstract

The need to condition distributional properties such as expectation, variance, and entropy arises in algorithmic fairness, model simplification, robustness and many other areas. At face value however, distributional properties are not random variables, and hence conditioning them is a semantic error and type error in probabilistic programming languages. On the other hand, distributional properties are contingent on other variables in the model, change in value when we observe more information, and hence in a precise sense are random variables too. In order to capture the uncertain over distributional properties, we introduce a probability construct – the random conditional distribution – and incorporate it into a probabilistic programming language Omega. A random conditional distribution is a higher-order random variable whose realizations are themselves conditional random variables. In Omega we extend distributional properties of random variables to random conditional distributions, such that for example while the expectation a real valued random variable is a real value, the expectation of a random conditional distribution is a distribution over expectations. As a consequence, it requires minimal syntax to encode inference problems over distributional properties, which so far have evaded treatment within probabilistic programming systems and probabilistic modeling in general. We demonstrate our approach case studies in algorithmic fairness and robustness.

Keywords Probabilistic programming, inference, modeling

1 Introduction

Probabilistic programming languages encode probabilistic models as programs. They are the most expressive among a long lineage of formalisms including Bayesian networks, factor graphs, and systems of statistical equations. A formalism is more expressive if it can represent a larger class of models or permits a wider range of inference queries. A class of more recent languages including Church [**?**], Anglican [**?**] and Venture [**?**] have tended towards one extreme of the spectrum, allowing recursion, control flow, and unbounded (or even random) numbers of discrete, continuous, or arbitrarily-typed random variables. These languages extend Turing complete deterministic languages to attain a notion of probabilistic universality: the ability to express any computable probability distribution.

There is a a blind spot – in terms of expressiveness – in even universal probabilistic programming languages. Existing probabilistic languages provide mechanisms to define random variables, transform them and condition them. Many languages also provide mechanisms to compute *distributional properties* such as expectation, variance, and entropy, or approximate them from samples. However, they lack any automated or composable mechanisms to capture the uncertainty over such distributional properties, which is a major limitation.

Distributional properties are fixed (often real) values, but in a sense they are random variables too. For example, rainfall depends on temperature, the season, the presence of clouds, and so on. Probabilistic models are routinely used to capture uncertainty over these factors and their complex interactions. With respect to a model, expected rainfall is a real value, but it changes if we obtain new information. For example it rises if we observe clouds and falls to zero if we observe their absence. These two expectations become a random variable over expectations – a *conditional expectation* – when we take into account the probabilities of the presence or absence of clouds. Moreover, for each random variable in the model there is a corresponding conditional expectation. For instance, with respect to the season, conditional expected rainfall is a random variable over four expectations, one for each season; with respect to temperature it is a continuous distribution. These conditional expectations capture the uncertainty over expected rainfall that results from other variables in the model, whereas the unconditional expected rainfall averages all the uncertainty away.

The ability to capture uncertainty over distributional properties is useful for the same reason that distributions are preferable to point estimates in general: they possess more

information. In addition, conditioning distributional properties opens up even more applications. For example, algorithmic fairness has received interest recently due to the expansion of machine learning into sensitive areas such as insurance, mortgages and employment. One probabilistic formulation is *equality of opportunity* [**?**], which states that the probability a qualified member in a minority group v_m receives a positive classification, is not far from the probability a qualified member in a majority group v_n receives a positive classification. These probabilities are distributional properties (probability is a special case of expectation). Capturing the uncertainty over them (randomizing them) allows us to condition a probabilistic classifier to satisfy equality of opportunity, rather than just verify whether it does or not.

Conditioning distributional properties is often computationally very challenging, but as inference procedures develop a number of problems could be unified under the same framework. For instance, classifications from deep neural networks have been shown to be vulnerable to adversarial attacks [**?**]. This could be mitigated by simply conditioning on the property that small random perturbations cannot dramatically change the classification distribution Another use case is to construct priors which adhere to distributional constraints. For example if we only have weak prior knowledge about a variable, such as its mean and variance, we can use a flexible nonparametric distribution family such as a Bayesian neural network (a neural network with a distribution over the weight parameters) but fix its mean and variance using random conditional distributions. Distributional properties may be functions of more than one random variable, for instance the Kullback-Leibler divergence and Wasserstein distance are functions of two random variables. Bounding divergences using random conditional distributions could be used for model simplification: to condition on a simple distribution being not far in divergence from a much more complex one.

As the primary contribution of this paper we introduce the *random conditional distribution* to randomize distributional properties. Given two random variables X and Θ, the random conditional distribution of X given Θ – which we denote $X \parallel \Theta$ – is a a *random distribution*: a random variable whose realizations are themselves random variables. In particular, each realization of $X \parallel \Theta$ is the random variable X conditioned on $\Theta = \theta$ where $\theta \sim \Theta$ is a realization of Θ.

Intuitively, the random conditional distribution decomposes a probabilistic model into a distribution over probabilistic models. For example, if $\Theta = \text{Bernoulli}(0.4)$ and $X = \mathcal{N}(\Theta, 1)$, then $X \parallel \Theta$ is a random conditional distribution comprised of two normally distributed random variables $\mathcal{N}(\Theta, 1) \mid \Theta = 0$ and $\mathcal{N}(\Theta, 1) \mid \Theta = 1$. The probabilities that $X \parallel \Theta$ takes these different outcomes is determined by the prior probabilities of the different outcomes of Θ: 0.4 and 0.6 respectively. We extend distributional properties to random conditional distributions such that while the expectation $\mathbb{E}(X)$ is a real value, $\mathbb{E}(X \parallel \Theta)$ is a distribution over real values (expectations). In particular, it is the distribution from taking the expectation of each of the two normally distributed random variables in $X \parallel \Theta$: 0 and 1 respectively.

The extension of distributional properties to random conditional distributions allows us to randomize distributional properties with minimal syntax. Returning to the fairness example, let $\text{offer}(v_m, \Theta)$ be a distribution over whether a candidate drawn from the distribution v_m will receive a job offer, where Θ is a distribution over weights of a probabilistic classifier. The classifier satisfies equality of opportunity if the ratio of probabilities of receiving an offer between groups is bounded by δ:

$$\frac{\mathbb{P}(\text{offer}(v_n, \Theta))}{\mathbb{P}(\text{offer}(v_m, \Theta))} < \delta \tag{1}$$

To construct a fair classifier rather than determine whether a classifier is fair, we want to condition on it being fair, i.e., find the conditional distribution over Θ given that Expression (1) holds. Again, this is problematic because Expression (1) is not a random variable and hence cannot be conditioned, despite being composed of random variable parts. Θ, v_m, and hence $\text{offer}(v_m, \Theta)$ are all random variables, but $\mathbb{P}(\text{offer}(v_m, \Theta))$ is a real value (the probability someone drawn from v_m receives a job offer), and Expression (1) is a Boolean – the classifier is either fair or not. In contrast, $\mathbb{P}(\text{offer}(v_m, \Theta) \parallel \Theta)$ is not a probability but a distribution over probabilities that v_m receives a job offer. Similarly, to turn expression 1 from a Boolean into a Boolean-valued random variable that can be conditioned requires only a small change:

$$\frac{\mathbb{P}(\text{offer}(v_n, \Theta) \parallel \Theta)}{\mathbb{P}(\text{offer}(v_m, \Theta) \parallel \Theta)} < \delta \tag{2}$$

While random conditional distributions can greatly improve the expressiveness of probabilistic programming, do we need new probabilistic languages, or can random conditional distributions be implemented in existing ones? We find that although one can explicitly define random conditional distributions for specific situations in existing languages, it is not possible to define a generic function $\parallel$ which automatically induces random conditional distributions from the original model. Based on this, we have designed a probabilistic programming language Omega for distributional inference using random conditional distributions.

In summary, we address the neglected problem of distributional inference: the randomizing and conditioning of distributional properties. Our method depends on the random conditional distribution, a higher-order probability construct. These concepts are synthesized in a probabilistic programming language Omega, which has an operator to construct random conditional distributions as a primitive construct. In more detail:

1. We formalize random conditional distributions (section 4) within measure theoretic probability.
2. We present the syntax and semantics of a probabilistic programming language Omega for distributional inference (section 5) using random conditional distributions.
3. Finally, we demonstrate Omega using several representative benchmarks including two case studies, where it is applied to infer a fair classifier and where it is applied to infer a classifier that is robust to adversarial inputs (section 6).

2 Background on Uncertain Probabilities

Probabilities over probabilities, often called higher-order probabilities [**?**], subsumed most of the inquiry into the interpretation of uncertain distributional properties. Interest in higher-order probabilities was motivated by an apparent failure of standard probability theory to distinguish between uncertainty and ignorance. For instance, a weather forecaster may project a 30% chance of of rain, which is straightforwardly captured as a first-order probabilistic statement: $\mathbb{P}(rain) = 0.3$. However, she may feel only 70% confident in that assessment. How to both express and interpret degrees of confidence in probabilistic statements within probability theory has led to much debate and disagreement.

Pearl [**?**] summarized several difficulties with higher-order probabilities. Traditional probability theory requires that probabilities are assigned to factual events that in principle could be verified by empirical tests. Under this principle it is problematic to assign probabilities to probability distributions themselves, because the truthfulness of probabilistic statements cannot be determined empirically. The statement $\mathbb{P}(rain) = 0.3$ could be verified, albeit with much difficulty, given sufficient knowledge of the physical systems which govern the weather. In contrast, the second order statement describing the weather forecaster's confidence, denoted by $\mathbb{P}\left[\mathbb{P}(rain) = 0.3\right] = 0.7$ (assuming momentarily this is syntactically valid), appears much less amenable to scrutiny by empirical test. What do probabilities of probabilities mean then, if their truthfulness cannot be determined empirically even in principle?

Several probabilistic [**? ?**] and non-probabilistic [**? ?**] formalisms have been developed to unify uncertainty and confidence. However, Pearl [**?**] and Kyburg [**?**] argued that standard first-order probability suffices; neither second-order nor higher-order extensions to probability theory were necessary. In particular, probabilistic statements of the form $\mathbb{P}(A) = p$ are themselves empirical events. To say such an event occurred means roughly to say one mentally computed that the probability of A is p. Hence events of this kind are subjective, and although not amenable to public scrutiny, are of no lesser stature than any other event.

The question of what renders a statement such as $\mathbb{P}(A) = p$ an unknown, random event, rather than a fixed outcome of a deterministic procedure, remains. A resolution first presented by de Finetti [**?**] suggested that $\mathbb{P}(A) = p$ is a random event whenever the assessment of $\mathbb{P}(A)$ depends substantially on other events in the system. For instance the assessment $\mathbb{P}(rain) = 0.3$ is uncertain because it depends substantially on $\mathbb{P}(clouds)$; the occurrence or non-occurrence of clouds would dramatically change our confidence in $\mathbb{P}(rain) = 0.3$. Pearl [**?**] formalized this notion of dependence within the framework of causal probabilistic models, asserting that an event A substantially depends on B if B is a cause of A with respect to a causal model. Crucially, Pearl demonstrated that the causal model provides both necessary and sufficient information for computing both $\mathbb{P}(A) = p$ and $\mathbb{P}\left[\mathbb{P}(A) = p\right]$. In other words, higher-order probabilities exist, but they are derived entirely from the original model.

Modern measure theoretic probability uniformly accommodates first and higher-order statements. In particular, since the probability of an event is the expectation of the random variable that indicates it, higher-order probabilities are conditional expectations. That is, uncertainty over a probability $\mathbb{P}(A)$ is captured by the conditional expectation $\mathbb{E}(\mathbf{1}_A \mid C)$ with respect to some contingency set C, where $\mathbf{1}_A(\omega)$ is 1 if $\omega \in A$ and 0 otherwise. Still, several issues remain. Namely, how to (i) generalize from higher-order probabilities to all distributional properties (ii) operationalize them within probabilistic programming languages so that they can be used in practice.

3 Background

In this section we introduce the foundations for our approach, which is largely measure-theoretic probability [**?**].

Random Variables. Probability models lie on top of probability spaces. A probability space is a measure space $(\Omega, \mathcal{H}, \mathcal{P})$, where Ω is the sample space, $\mathcal{H}$ is a σ-algebra over subsets of Ω, and $\mathcal{P}$ is the probability measure ($\mathcal{P}(\Omega) = 1$). Random variables are functions from the sample space Ω to a realization space τ. A model is a collection of random variables along with a probability space. A concrete example of a probability space takes Ω to be hypercube, with $\mathcal{P}$ being uniform a uniform measure over that hypercube. A normally distributed random variable maps from $\Omega \to \mathbb{R}$. Since the underlying probability space is uniform, this function is the inverse cumulative distribution function of the normal.

Random Variable Algebra It is convenient to treat random variables (which are functions) as if they were values from their realization space. For example, if X is a random variable, then $X + X$ and $X/3$ are also random variables. The semantics of this syntax defines operations on random variables pointwise. For example: $(X + X)(\omega) = X(\omega) + X(\omega)$. More generally, let $X : \Omega \to \tau_1$ and $f : \tau_1 \to \tau_2$ be a

function, then $Y = f(X)$ is a random variable defined as:

$$Y(\omega) = f(X(\omega)) \tag{3}$$

Conditioning Conditioning restricts a model to be consistent with a predicate. It can be operationalized as an operator | that concentrates the probability space of a random variable $X : \Omega \to \tau$ to an event A indicated by a predicate Y, i.e.: $A = \{\omega \mid Y(\omega) = 1\}$. That is, $X \mid Y : \Omega \to \tau$ is functionally identical to $X - (X \mid Y)(\omega) = X(\omega)$ – but defined on the probability space $(\Omega \cap A, \{A \cap B \mid B \in \mathcal{H}\}, \mathcal{P}/\mathcal{P}(A))$, the concentration of $\mathcal{S}$ onto A.

The general construction of new models might require conditioning on sets of measure zero. This process can be made rigorous via disintegration [**?**], which can be thought of as the reversal of building joint distributions through product measure constructions.

Distributional Properties and Operators Distributional properties such as expectation, variance and entropy summarize aspects of a random variable. We use the term distributional operator for the higher-order function that maps random variables to the corresponding distributional property. For example the expectation distributional operator $\mathbb{E} : (\Omega \to \mathbb{R}) \to \mathbb{R}$ maps a random variable X to its expectation $\mathbb{E}(X)$. It is rigorously defined in terms of Lebesgue integration over the sample space Ω with respect to the probability measure $\mathcal{P}$:

$$\mathbb{E}(X) = \int_{\Omega} X(\omega)\, d\mathcal{P}(\omega)$$

Distributional properties are in many cases intractable to compute exactly, but can be approximated from samples.

4 The Random Conditional Distribution

Random conditional distributions provide a mechanism to condition distributional properties. Given two random variables X and Θ, the random conditional distribution of X given Θ – which we denote $X \parallel \Theta$ – is a a *random distribution*: a random variable whose realizations are themselves random variables. In particular, each realization of $X \parallel \Theta$ is the random variable X conditioned on $\Theta = \theta$ where $\theta \sim \Theta$ is a realization of Θ:

Definition 1. The random conditional distribution (rcd) of a random variable $X : \Omega \to \tau_1$ given $\Theta : \Omega \to \tau_2$ is a random variable $X \parallel \Theta : \Omega \to (\Omega \to \tau_1)$, defined as:

$$(X \parallel \Theta)(\omega) := X \mid \Theta = \Theta(\omega) \tag{4}$$

Intuitively, the random conditional distribution decomposes a probabilistic model into a distribution over probabilistic models. For example, if $\Theta = \text{Bernoulli}(0.4)$ and $X = \mathcal{N}(\Theta, 1)$, then $X \parallel \Theta$ is a random conditional distribution comprised of two normally distributed random variables $\mathcal{N}(\Theta, 1) \mid \Theta = 0$ and $\mathcal{N}(\Theta, 1) \mid \Theta = 1$. The probabilities that $X \parallel \Theta$ takes these different outcomes is determined by the prior probabilities of the different outcomes of Θ: 0.4 and 0.6 respectively.

A consequence of random variable algebra (section 3) is that applying distributional operators to random condition distributions yields random distributional properties. Continuing the example from above, $\mathbb{E}(X \parallel \Theta)$ is a random variable over expectations taking values 0 and 1 with probabilities 0.4 and 0.6 respectively. $\mathbb{E}(X \parallel \Theta)$ is a conditional expectation, denoted $\mathbb{E}(X \mid \Theta)$ in standard mathematical notation.

Theorem 1. $\mathbb{E}(X \parallel \Theta)$ is the conditional expectation of X with respect to Θ defined as $\mathbb{E}(X \mid \sigma(\Theta))(\omega) = \mathbb{E}(X \mid \Theta = \Theta(\omega))$.

Proof. For clarity we distinguish expectation defined on real valued random variables $\mathbb{E}$ from expectation defined on real valued random distributions $\tilde{\mathbb{E}}$:

$$\tilde{\mathbb{E}}(X \parallel \Theta) = \tilde{\mathbb{E}}(\lambda\omega.(X \mid \Theta = \Theta(\omega))) = \lambda\omega.\, \mathbb{E}(X \mid \Theta = \Theta(\omega))$$

□

The expectation of a random conditional distribution is a conditional expectation, but the same mechanism works for any distributional property. If $\mathbb{O}$ is a distributional operator defined on τ valued random variables, then it extends pointwise to random conditional distributions:

$$\mathbb{O}(X \parallel \Theta)(\omega) = \mathbb{O}((X \parallel \Theta)(\omega)) = \mathbb{O}(X \mid \Theta = \Theta(\omega))$$

$\mathbb{O}$ could be expectation, variance, entropy, information, or support. $\mathbb{O}$ could also map from more than one random variable, such as KL-Divergence or mutual information.

It is possible that uncertainty over a distributional property is captured by a variable already in the model, which is simpler than our construction using random conditional distributions. For example, if $\Theta = \text{Uniform}(0, 1)$ and $X = \text{Bernoulli}(\Theta)$, then Θ is equal to the conditional expectation $\mathbb{E}(X \mid \Theta)$, and therefore equal to $\mathbb{E}(X \parallel \Theta)$. This is a consequence of the fact that the weight parameter of a Bernoulli distribution is also its expectation.

However, this is a special case; for most distributional properties, most models do not possess a corresponding variable. For example the variance of X is not a variable in example model above, and if we substitute the $\text{Bernoulli}(\Theta)$ with $\text{Beta}(\Theta, 1)$, neither is the expectation. In some cases a distributional property that is not in the model can be derived as a transformation of existing variables. Continuing the example, if $X = \text{Beta}(\Theta, 1)$, then its expectation is the random variable $\Theta/(\Theta + 1)$. This however, is also not always possible. In many models there exists no closed form expression for a given distributional property. For example if we instead have $X = \text{Gamma}(\Theta, 1)$, there is no closed form expression from the median.

Even when it is possible, a further limitation of this approach is that it lacks the flexibility afforded by the second

$$
\begin{array}{lll}
\text{type} & \tau ::= & \text{Int} \mid \text{Bool} \mid \text{Real} \mid \Omega \mid \text{RV}\,\tau \mid \tau_1 \to \tau_2 \mid (\tau_1, \tau_2) \\
\text{term} & t ::= & n \mid b \mid r \mid \bot \mid x \mid \text{ if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1(t_2) \mid \\
 & & (t_1, t_2) \mid \text{let } x = t_1 \text{ in } t_2 \mid \lambda x : \tau . t \mid \\
 & & \text{rv}(t) \mid \text{ciid}(t) \mid \text{uid} \mid (t_1 \mid t_2) \mid t_1 \parallel t_2 \mid \text{dist}(t) \\
\text{query} & & \text{rand}(t)
\end{array}
$$

Figure 1. Abstract Syntax for Omega

argument of rcd. The uncertainty in $\mathbb{O}(X \parallel \Theta)$ is a consequence of uncertainty in Θ. Changing Θ reveals different perspectives and granularities of the uncertainty. For example, if deciding when to take a mountaineering trip one might be interested in $\mathbb{E}(rain \parallel season)$, but if deciding how tall of a mountain to tackle then $\mathbb{E}(rain \parallel altitude)$ may be more informative. Moreover Θ can be crucial when conditioning distributional properties. In the equality of opportunity example from the introduction, if we instead find the conditional distribution over Θ given that

$$\frac{\mathbb{P}(\text{offer}(v_n, \Theta) \parallel (\Theta, v_n, v_m))}{\mathbb{P}(\text{offer}(v_m, \Theta) \parallel (\Theta, v_n, v_m))} < \delta \tag{5}$$

holds, we would in effect eliminate portions of the population. This is clearly not what is meant by fairness.

5 The Omega Programming Language

This section describes the core language of Omega: a functional language augmented with a small number of probability constructs. Most of the probabilistic constructs – random variables, probability spaces and conditional independence – correspond directly to concepts in measure-theoretic probability. Figure 1 shows the abstract syntax of Omega.

5.1 Types

The type system is composed of primitives Int, Bool, and Real with their standard interpretation. There are two primitive probabilistic types: Ω the sample space type, and $\text{RV}\,\tau$ the τ-valued random variable type. Function types are expressed with $\tau \to \tau$. Omega is curried, and hence multivariate functions have nested function types of the form $\tau \to \cdots \to \tau$. Tuple types are denoted with (t_1, t_2). Every type is lifted to include $\bot$, the undefined value, which plays an important role in conditioning. Typing rules on probabilistic terms are given in figure 3.

5.2 Syntax

Standard terms have their standard interpretation. n represents integer numbers, b are Boolean values in $\{\text{true}, \text{false}\}$, and r are real numbers. x represents a variable in a set of variable names $\{\omega, x, y, z, \dots\}$. $\bot$ represents the undefined value. $\lambda x : \tau . t$ is a lambda abstraction; $t_1(t_2)$ is function application; conditionals are expressed with if t_1 then t_2 else t_3; variable bindings are defined with let.

Macros:

uid = integer unique across program

Functions:

x | y = x:RV , x:RV Bool .
rv(:Ω.if y( ) then x( ) else)
x || θ = x:RV 1, :RV 2 . :Ω.x | = ( )

Figure 2. Built-in functions and macros in Omega

Omega has several probabilistic terms. If t is a function of type $\Omega \to \tau$, $\text{rv}(t)$ is a random variable. $\text{ciid}(t)$ creates a new random variable that is distributed identically to t, but is conditionally independent given parent random variables it depends on. uid is a macro (i.e. resolved syntactically) that evaluates unique integer. It is is used to extract a unique element of the sample space. $t_1 \mid t_2$ constructs a conditional random variable equivalent to t_1 but defined on a sample space restricted by t_2. The random conditional distribution of t_1 given t_2 is denoted by $t_1 \parallel t_2$. The query term $\text{rand}(t)$ draws a sample from a random variable t.

Operators | and || map from and to arbitrarily-valued random variables, and hence are higher-order functions with polymorphic types. For simplicity of presentation Omega lacks user-defined polymorphic types, but we implement these built-in polymorphic functions (figure 2).

Operator dist represents broadly functions that evaluate distributional properties of random variables. Its type is RV T → Real where T is a primitive type. We rely on the underlying inference engine to provide these functions and treat dist as a blackbox in this section. In particular, we focus on functions that are deterministic and can be approximated using sampling, such as expectation, variance, and KL divergence.

$$\frac{\Gamma \vdash t_1 : \Omega \to \tau}{\Gamma \vdash \text{rv}(t_1) : \text{RV}\,\tau}\ rv \qquad \frac{\Gamma \vdash t_1 : \text{RV}\,\tau}{\Gamma \vdash \text{ciid}(t_1) : \text{RV}\,\tau}\ ciid$$

$$\frac{\Gamma \vdash t_1 : \text{RV}\,\tau \quad \Gamma \vdash t_2 : \text{RV}\,\text{Bool}}{\Gamma \vdash t_1 \mid t_2 : \text{RV}\,\tau}\ cond$$

$$\frac{\Gamma \vdash t_1 : \text{RV}\,\tau_1 \quad \Gamma \vdash t_2 : \text{RV}\,\tau_2}{\Gamma \vdash t_1 \parallel t_2 : \text{RV}\,\text{RV}\,\tau_1}\ rcd$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \text{RV}\,\tau_1}{\Gamma \vdash (t_1\ t_2) : \text{RV}\,\tau_2}\ rvapp$$

$$\frac{\Gamma \vdash t : \text{RV}\,\tau}{\Gamma \vdash \text{rand}(t) : \tau}\ rand$$

Figure 3. Typing rules of Omega

5.3 Denotational Semantics

Here we define a denotational semantics of Omega which is shown in figure 4. In the Appendix, we provide an operational semantics, which is more low-level and closer to our implementation. In the denotational semantics, standard terms

are standard and omitted. The denotation $[\![t]\!]$ of a term t is a value in a semantic domain corresponding to an Omega type, such as a Boolean, real number, or random variable.

Semantic Domains

For the purpose of the denotational semantics, we assume an unconditioned probability space $\mathcal{S}^{\mathrm{U}} = (\Omega, \mathcal{H}, \mathcal{P})$, Ω is a d-dimensional unit hypercube $[0,1]^d$:

$$\Omega = \Omega_1 \times \Omega_2 \times \cdots \times \Omega_d \text{ where } \Omega_i = [0,1]$$

If $\omega \in \Omega$, then ω_i denotes the ith component and is a real value in $[0,1]$. $\mathcal{P}$ is any probability measure such that the set of random variables $\{\lambda\omega : \Omega.\omega_i \mid \forall i\}$ are mutually independent and uniformly distributed on $[0,1]$.

$\mathcal{S}^{\mathrm{U}}$ is natural choice of a probability space because any univariate probability distributions can be constructed by transformation of a uniform distribution over the unit interval, and arbitrary multivariate random variables can be defined by transformations of the unit hypercube.

Different random variables may be conditioned on different events, and hence defined on different probability spaces. Specifically, different random variables may be defined on different concentrations of the unrestricted space $\mathcal{S}^{\mathrm{U}}$. If A is some event of positive measure, $\mathcal{S}' = \mathcal{S}^{\mathrm{U}} \cap A$ is the concentration of $\mathcal{S}^{\mathrm{U}}$ onto A:

$$\mathcal{S}^{\mathrm{U}} \cap A = (\Omega \cap A, \{A \cap B \mid B \in \mathcal{H}\}, \mathcal{P}/\mathcal{P}(A))$$

Random variables exist in a parent-child relation. This relation has a causal interpretation, functional realization, and probabilistic consequences. If Z is the sole parent of X and Y, then (i) Z represents a generative process whose outcome causes the outcomes of X and Y, (ii) the evaluation of $X(\omega)$ (or $Y(\omega)$) requires the evaluation of $Z(\omega)$, but not vice-versa, and (iii) X and Y are conditionally independent given Z. A particularly common relationship pattern is a class of random variables that are mutually conditionally independent and identically distributed (c.i.i.d) given their parents. This arises when an experiment is repeated (e.g. a die is tossed several times), or multiple noisy observations of the same process are taken. As such, unique elements of a c.i.i.d. classes are the primitive representation of probability distributions in Omega.

Definition 2. The c.i.i.d. class of X given Z is denoted $[X]_{\perp\!\!\!\perp|Z}$ and inductively defined by:

1. $X \in [X]_{\perp\!\!\!\perp|Z}$
2. $A \in [X]_{\perp\!\!\!\perp|Z}$ if for all $B \in [X]_{\perp\!\!\!\perp|Z}$ where $A \neq B$:
   (i) A is independent of B given Z
   (ii) A is identically distributed to B given Z

To represent a random variable as a unique member of a c.i.i.d. class, we first observe that two random variables are c.i.i.d. given a third if they apply the same transformation to disjoint component of the sample space. For example, $X(\omega) = \omega_1 + Z(\omega)$ and $Y(\omega) = \omega_2 + Z(\omega)$ are c.i.i.d. given $Z(\omega) = \omega_3$ since if Z is fixed (i) they map from disjoint components of Ω (they are conditionally independent), and (ii) the mapping is functionally identical (they are identically distributed).

Rather than *syntactically* express that two random variables map from disjoint components of Ω, we enforce it semantically by assigning each random variable X a unique set of the dimensions of Ω. That is, we define X as a function of a *projection* of Ω – characterized by an index set I_X – from a collection of unique projections: $((\Omega_1 \times \Omega_2, \times \cdots), (\Omega_j \times \Omega_{j+1} \times \cdots), \ldots)$. For X to be a defined on a projection means that any application $X(\omega)$ is replaced with $X(\omega_{I_X})$ where $\omega_{I_X} = \mathrm{proj}_{I_X}(\omega) = (\omega_i, \omega_j, \omega_k, \ldots)$ for all $i, j, k, \ldots \in I_X$.

For the same reasons as above, two random variables will then be i.i.d. if they apply the same transformation to different projections of Ω. For example, if $f(\omega) = \omega_1 + \omega_2$, then $X(\omega) = f(\mathrm{proj}_{I_X}(\omega))$ and $Y(\omega) = f(\mathrm{proj}_{I_Y}(\omega))$, are i.i.d. if I_X, I_Y are disjoint sets.

Projection is complicated by the fact that random variables typically have parents. If Z is the parent of X, e.g. in $X(\omega) = \omega_1 + Z(\omega)$, then the evaluation of $X(\omega)$ necessitates the evaluation of $Z(\omega)$. Both X and Z are defined on their respective projections I_X and I_Z, but if we first project ω onto I_X to evaluate $X(\omega)$, we discard the elements necessary to project onto I_Z in the evaluation $Z(\omega)$ within X.

To resolve this, we represent an element of the sample space as a reversible projection $\omega^\pi = (\omega, I)$, where $\omega \in \Omega$ is an element of the unprojected sample space, and I is an index set. $\omega^\pi_i = (\mathrm{proj}_I(\omega))_i$ is the ith element of the projection onto I. Crucially, ω^π can be projected from I onto another index set J trivially by substituting I with J.

We represent, a random variable X as (f_X, I_X), where $f_X : \Omega \to \tau$ is a function and I_X is an index set of natural numbers. X is defined on a (potentially) concentrated probability space denoted $\mathcal{S}(X) = \mathcal{S}^{\mathrm{U}} \cap \{\omega \mid X(\omega) \neq \perp, \omega \in \Omega\}$. Random variable application reprojects ω onto I_X:

$$X(\omega) := f_X(\mathrm{proj}_{I_X}(\omega))$$

Denotations

The operator rv pairs a function of Ω (constructed using lambda abstraction) with a probability space to yield a random variable. For example: $[\![\mathrm{rv}(\lambda\omega : \Omega.0)]\!]$ denotes the constant random variable $(\omega \mapsto 0, I)$.

For a random variable to not be constant, it must access the sample space. $[\![\omega(i)]\!]$ (rule ω*app*) denotes an element in $[0,1]$ of the sample space. In particular, it is the ith element of a projection of Ω. By rule rvapp, this is the projection of the random variable that the application $\omega(i)$ is within. Since different random variables are defined on different projections, $\omega(i)$ within the context of distinct random variables are independent. The value of i is arbitrary, but indices must not be reused if independent values are needed. The macro uid resolves to a globally unique integer index, which

relieves the programmer from specifying arbitrary indices. For example, $[\![\mathrm{rv}(\lambda\omega : \Omega.\omega(\mathrm{uid}) + \omega(\mathrm{uid}))]\!]$ denotes the random variable $(\omega \mapsto \omega_1 + \omega_2, I)$.

Random variables can be treated like values from their realization space using pointwise random variable (rule pointwise) composition, e.g.: $[\![\mathrm{sqrt}(\mathrm{rv}(\lambda\omega : \Omega.\omega(\mathrm{uid}))]\!]$ denotes the random variable $(\omega \mapsto \sqrt{\omega_1}, I)$.

$[\![\mathrm{ciid}(t)]\!]$ is functionally identical to $[\![t]\!]$ but on a different projection of Ω, and hence c.i.i.d. given the parents of $[\![t]\!]$. For example, if $\mathrm{unif} = \lambda\omega : \Omega.\omega(\mathrm{uid})$, then $[\![\mathrm{unif} + \mathrm{unif}]\!]$ is a equivalent to $[\![2 * \mathrm{unif}]\!]$, whereas $[\![\mathrm{ciid}(\mathrm{unif}) + \mathrm{ciid}(\mathrm{unif})]\!]$ is a sum of independent uniforms, a triangular distribution.

rand samples from a distribution. In rule (rand), $\omega \sim \mathcal{S}'$ denotes that ω is an i.i.d. sample drawn from the probability space $\mathcal{S}'$. That is, we assume the existence of an oracle able to randomly select $\omega \in \Omega'$ with probability $\mathcal{P}'(\{\omega\})$. For example $[\![\mathrm{rand}(\mathrm{rv}(\lambda\omega : \Omega.\omega(\mathrm{uid}) > 0.4))]\!]$ denotes either true or false with probabilities 0.6 and 0.4 respectively.

$[\![t_1 \mid t_2]\!]$ denotes a random variable $[\![t_1]\!]$ conditioned on $[\![t_2]\!]$ being true by concentrating the probability space of $[\![t_1]\!]$ onto the event indicated by $[\![t_2]\!]$ (rule cond). It is realized by a built-in function, which returns a random variable which maps ω to $\perp$ if $[\![t_2]\!](\omega) = 0$. For example, if X is a normally distributed random variable then $[\![X \mid X > 0]\!]$ is a truncated normal defined on a concentrated space $\mathcal{S}'$.

The random conditional distribution operator || is defined in terms of | and lambda abstraction as a built-in function (figure 2).

$[\![\mathrm{dist}(t)]\!]$ calculates the distributional property of random variable t. There are two cases depending on the type of t. When t is a regular random variable whose realization space is primitive types, we rely on the underlying inference engine (denoted by $[\![\mathrm{dist}]\!]$) to return the desired property (rule (dist)). For the specific engine we use, it approximates these properties by sampling. When t is a higher-order random variable whose realization space is random variables, rule pointwise would be triggered if dist was a regular function. However, it is not so rule (higher-order dist) is defined to provide for a similar effect. It pushes dist inside the definition of t recursively until it reaches the inner-most random variable. This effectively creates distributions of distributional properties.

We now show that a random variable defined in Omega represents a well-defined probability distribution. To simplify the discussion, without reducing the expressiveness, we do not allow rand to be invoked inside a function f that is used to construct a random variable ($\mathrm{rv}(f)$). We further limit the discussion to function f that always terminates.

Definition 3 (Well-formed randvars)**.** We say a random variable $t = \mathrm{rv}(f)$ is well-formed, if for any $\omega \in \Omega$, $f(\omega)$ terminates and rand is not invoked when evaluating $f(\omega)$.

Theorem 2 (Well-definedness)**.** The distribution represented by a well-formed random variable is well-defined.

$$[\![\mathrm{rv}(t)]\!] \mapsto ([\![t]\!], I) \text{ where } I \text{ is unique} \quad \text{(rv)}$$

$$[\![\mathrm{ciid}(t)]\!] \mapsto (f, J) \quad \text{(ciid)}$$
where $[\![t]\!] = (f, I)$ and J is unique

$$[\![t_1(t_2)]\!] \mapsto (\mathrm{proj}_I(\omega))_i \quad (\omega\text{app})$$
if $[\![t_1]\!] \mapsto (\omega, I)$ and $[\![t_2]\!] \mapsto i$

$$[\![t_1(t_2)]\!] \mapsto [\![e/(\omega, I))]\!] \text{ if } [\![t_1]\!] \mapsto (\lambda x : \Omega.e, I)$$
and $[\![t_2]\!] \mapsto (\omega, J)$ (rvapp)

$$[\![t_1(t_2)]\!] \mapsto (\lambda\omega : \Omega.g(f_X(\omega)), I) \quad \text{(pointwise)}$$
if $[\![t_1]\!] \mapsto g : \tau_1 \mapsto \tau_2$
and $[\![t_2]\!] \mapsto (f_X : \Omega \mapsto \tau_1, I)$

$$[\![\mathrm{rand}(t)]\!] \mapsto [\![f((\omega, N))]\!] \text{ where } \omega \sim \mathcal{S}((f, I)) \quad \text{(rand)}$$
if $t \mapsto (f, I)$

$$[\![\mathrm{dist}(t)]\!] \mapsto [\![\mathrm{dist}]\!]((f, I)) \text{ if } [\![t]\!] \mapsto (f : \Omega \mapsto T, I) \quad \text{(dist)}$$
where T is Bool, Real, or Int

$$[\![\mathrm{dist}(t)]\!] \mapsto (\lambda\omega : \Omega.[\![\mathrm{dist}(f(\omega))]\!], I) \quad \text{(higher-order dist)}$$
if $[\![t]\!] \mapsto (f : \Omega \mapsto \mathrm{RV}\,\tau, I)$

Figure 4. Denotational Semantics

Proof. Assume the variable is $t \mapsto (f, I)$. We prove the theorem by showing the probability density function/probability mass function of t is well-defined. For simplicity, we assume t is continuous and the proof for the case where t is discrete is similar. According to (rand) rule, rand will only draw ω that falls into the domain of f:

$$\omega \sim \mathcal{S}((f, I)) = \mathcal{S}^{\mathrm{U}} \cap \{\omega \mid (f, I)(\omega) \neq \perp, \omega \in \Omega\}.$$

Then the range of $\mathrm{rand}(t)$ is $R = \{(f, I)(\omega) \mid \omega \in \Omega \wedge (f, I)(\omega) \neq \perp\}$. Let pdf be the pdf of t and $V = \mathrm{Bool} \cup \mathrm{Int} \cup \mathrm{Real} \cup (\mathrm{RV}\,\tau)$. For $v \in V \setminus R$, we have $pdf(v) = 0$. For $v \in V \cap R$, its support is $D = \{\omega \mid (f, I)(\omega) = v, \omega \in \Omega\}$. Since t is well-formed, the evaluation of $(f, I)(\omega)$ is deterministic and always terminates. Recall that $\mathcal{S}^{\mathrm{U}}$ is a uniform distribution on d-dimensional unit Ω, then we have $pdf(v) = \sum_{\omega \in D}(\frac{1}{1-0})^d$. Thus, pdf is well-defined over V. □

5.4 Example Omega Programs

Here we demonstrate Omega through examples.

Sample from a standard uniform distribution:

```
let x = rv( λω: Ω.ω(uid)) in rand(x)
```

Distribution families map parameters to random variables:

```
bern = λp: Real.rv( λω: Ω.ω(uid) > p)
uniform = λa: Real, b: Real.rv( λω: Ω. (uid)*(b-a)+
```

Pass random variables as parameters to a family for Bayesian parameter estimation:

```
let μ = uniform(0, 1)
   x1 = uniform(μ, 2)
   x2 = uniform(μ, 2) in
   rand(μ | rand(x1 = 1.3) ∧ rand(x2 = 1.6))
```

Variance is expressible in terms of $\mathbb{E}$:

```
var = λx : RV Real.E((x - E(x))^2)
```

Laws of total expectation and variation. Given x and y: Variance is expressible in terms of $\mathbb{E}$:

```
E(x) == E(E(x || y))
var(y) == E(var(y || x)) + var(E(y || x))
```

5.5 Inference

Here we outline the general interface to inference in Omega

A conditional sampling algorithm in Omega is an implementation of the procedure rand.

Definition 4. A conditional sampling algorithm is function $\text{rand} : (\Omega \to \tau) \to \tau$ which maps a random variable to a value sampled from its domain.

Rejection Sampling rand_{rej} draws exact samples by rejecting those violating specified conditions. It assumes a source of randomness $\mathcal{A}$: an infinite matrix where $A_{i,j} \in [0,1]$ is randomly and uniformly selected by an oracle. A function $\text{rand}_\omega(j) = \omega$ where $\omega_i = \mathcal{A}(i,j)$ maps integers to independent sample space elements. rand_{rej} is defined recursively:

$$\text{rand}_{\text{rej}}(x) = \text{rand}_{\text{rej}}(x, 1)$$

$$\text{rand}_{\text{rej}}(x, j) = \begin{cases} x(v) & \text{if } x(v) \neq \bot \\ \text{rand}_{\text{rej}}(x, j+1) & \text{otherwise} \end{cases}$$

$$\text{where } v = x(\text{rand}_\omega(j))$$

Constraint Relaxation The expected time to draw a sample with rejection sampling is inversely proportional to the measure of the conditioning set, which can be vanishingly small. Instead, most probabilistic programming languages apply more sophisticated inference methods to sample from posterior densities derived automatically from the model (e.g. by [?]). However, often this is inapplicable to distributional inference problems because the likelihood terms are intractable. Likelihood-free inference procedures are then the sole option.

We use a recent likelihood-free approach given in [?]. From a predicate Y it constructs an energy function $U_Y : \Omega \to \mathbb{R}$ which approximates Y. It relies on a distance metric, such that regions closer to the event indicated by Y become more probable. U_Y is then sampled from using a variant of replica exchange Markov Chain Monte Carlo. We defer to [?] for more details.

6 Evaluation

We first compare the expressiveness of Omega against an existing probabilistic language using a representative benchmark suite both qualitatively and quantitatively. Then we show that Omega is able to enable emerging applications using two case studies.

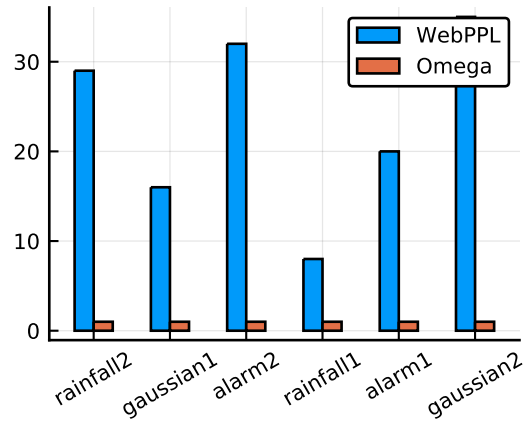

Figure 5. Empirical comparison of lines of code

6.1 Expressiveness

Random conditional distributions increase the expressiveness of probabilistic programming. Here we support this claim by comparison with WebPPL. In particular, we show that while it is possible to express rcd queries in WebPPL by carefully constructing the model, it requires substantial effort to change the program to express queries that are different from the original ones. In contrast, the effort to achieve similar effect in Omega is much less.

6.1.1 Qualitative Analysis

Consider an Omega program that samples from the conditional expectation $\mathbb{E}(\textit{rain} \mid \textit{iswinter})$:

Pass random variables as parameters to a family for Bayesian parameter estimation:

```
let bern = λp . λω: Ω.ω(uid) > p
  iswinter = bern(0.5)
  clouds = if iswinter then bern(0.8) else bern(0.3
  base = if iswinter then 3 else 0
  altitude = uniform([base + 3, base + 5, 10])
  rainfall = if clouds then altitude else 0 in
    mean(rainfall || clouds)
```

Here, mean approximates the expectation of rainfall by sampling. To sample a different distributional property, e.g. E(rainfall || iswinter) or var(rainfall || clouds), we only need to change the final query expression without touching the original model. Contrast this with the same example in WebPPL:

```
var model = function() {
  iswinter = bernoulli(.5)
  rainfall = Infer(function(){
    clouds = iswinter ? bernoulli(0.8) :
bernoulli(0.3)
    base = iswinter ? 3 : 0
    altitude = uniformDraw([3 + base, 5 + base, 10]
    return clouds ? altitude : 0.0
```

```
    })
  return expectation(rainfall)
}
```

By excluding iswinter within the definition of rainfall, we have split the model into two. A single sample from model will sample iswinter once, and with this value fixed will sample clouds, altitude, and base several times to approximate the expectation. This achieves the same effect as rcd.

The WebPPL model is ostensibly of similar complexity to the Omega model. However, we have had to bake the conditional expectation into the generative model. This violates the principle discussed in section 2 that distributional properties are derived from, not in addition to the original base model. We explicitly constructed a model split into two parts whereas || can construct any such split automatically. Of practical importance, changing the distributional property can require substantial changes to the model.

For example, to sample instead from $\mathbb{E}(rain \mid clouds)$ in Omega is trivial, whereas in WebPPL the model changes dramatically:

```
var winterandclouds = function()
{
  var iswinter = bernoulli(.5)
  var clouds = iswinter ? bernoulli(0.8) :
bernoulli(0.3)
  return {iswinter: iswinter, clouds: clouds}
}
var conditionedmodel = function(iscloudy)
{
  wandy = winterandylouds()
  iswinter = wandy.iswinter
  clouds = wandy.clouds
  condition(clouds == iscloudy)
  base = iswinter ? 3 : 0
  altitude = uniformDraw([3 + base, 5 + base, 10])
  return clouds ? altitude : 0.0
}
var model = function() {
  wandy = winterandylouds()
  iswinter = wandy.iswinter
  clouds = wandy.clouds
  rainfall = Infer(function(){
    return conditionedmodel(clouds)
    })
  return expectation(rainfall)
}
```

We must then manually construct a nested inference problem to sample the distributional property.

6.1.2 Quantitative Comparison

To evaluate this quantitatively, we evaluated the degree to which several models must be modified to construct random distributional proprties. For model $\mathcal{M}$ from a set of models (below), we:

1. Select two variables at random $\Theta, X \in \mathcal{M}$
2. From $\mathcal{M}$ manually construct $\mathcal{M}'$ where $X \parallel \Theta \in \mathcal{M}'$
3. Evaluate the number of lines from $\mathcal{M}$ to $\mathcal{M}'$

Models:

- *rainfall*: as above
- *alarm*: a model over alarm clock and time
- *gauss*: a binomial-gaussian model

Figure 5 demonstrates expressiveness improvements in Omega vs WebPPL. WebPPL requires a large number of edits for every change, whereas Omega requires exactly 1.

6.2 Case Studies

This section explores two studies of distributional inference. All inferences were performed on a Linux 4.15 laptop with a 1.9GHZ quad-core I7 processor and 24GB memory.

6.2.1 Inferring Fair Classifiers

In this case study, we aim to transfer an unfair non-probabilistic classifier into a fair one.

Setup. The classifier is studied by FairSquare [**?**], a tool for verifying algorithmic fairness. It is an SVM (SVM_4 in Section 6 of [**?**]) that predicts whether a person has a high income based on their gender, age, capital gains, and capital losses. The classifier is trained on a popular income dataset [1] and judged as unfair by FairSquare. Same as the setting in [**?**], we use equality of opportunity [**?**] as the fairness specification:

$$\frac{\mathbb{P}(high\ income \mid female \wedge age > 18)}{\mathbb{P}(high\ income \mid male \wedge age > 18)} > 0.85. \qquad (6)$$

Probabilistic Program. Figure 6 outlines the Omega program which infers parameters which make the SVM fair. Given fairness is a distributional property, we begin with constructing random variables which represent the input distribution by invoking popModel(). To implement popModel, we use a Bayesian network that is described in [**?**]. Then we construct a distribution of SVM parameters. While our end goal is to make the SVM fair, we also do not want its accuracy to degrade much. Therefore, we create an array of normals as the random parameters whose means are the original learned non-random parameters. This will make our final samples of parameters similar to the original ones. Next, we construct the fairness specification and lift it as a distribution which depends on the random parameters. Here we use prob to evaluate the probability of whether a statement holds, which is approximated by taking the mean of samples drawn from it. Finally, we draw samples from the random parameters conditioning on the fairness specification.

[1] https://archive.ics.uci.edu/ml/datasets/Adult

```
# input distribution
let gender, age, cap_gain, cap_loss = popModel() in
    # parameter distribution
    rparams = [normal(p, 1.0) for p in params] in

# output distribution
let high_income = SVM(rparams, gender, age,
                      cap_gain, cap_loss) in

# fairness specification
let f_high_income = cond(high_income,
                    gender == female and age > 18)
    m_high_income = cond(high_income,
                    gender == male and age > 18) in

let fairness = (prob(qual_female_high_income || rparams)
     / prob(qualμale_high_income || rparams)> 0.85) in

# parameter distribution conditioning on being fair
let fair_rparams = cond(rparams, fairness) in

# draw fair parameter samples
rand(fair_rparams)
```

Figure 6. Probabilistic program for inferring fair classifiers.

```
# input distribution
let gender, age, num_edu, cap_gain = popModel()
    # parameter distribution
    rparams = [normal(p, 1.0) for p in params] in

# output distribution
let high_income = NN(rparams, gender, age, num_edu,
                     cap_gain) in

# craft adversarial attack
let perturb = fast_gradient_sign(rparams, gender, ag
                    num_edu, cap_gain) in

let adv_gender, advαge, adv_num_edu, advγap_gain =
        gender+perturb[1], age+perturb[2],
    num_edu+perturb[3], cap_gain+perturb[4] in

# robustness specification
let adv_high_income = NN(rparams, adv_gender, advαge
                    adv_num_edu, advγap_gain) in

let robustness = (prob((adv_high_income == high_inco
                    ||
    rparams) > 0.99) in

# parameter distribution conditioning on being robu
let robust_rparams = cond(rparams, robustness) in

# draw robust parameter samples
rand(robust_rparams)
```

Figure 7. Probabilistic program for inferring robust classifiers.

Accuracy and efficiency. To validate our approach, we use FairSquare [**?**] to verify whether the produced SVM is fair. In theory, our result can be inaccurate due to two reasons: we approximate prob by samples, and the underlying solver itself does approximate inference. However, in this application, we observe running our program produces high-quality results: we sampled 10 parameters using each algorithm, and FairSquare returned fair for all of them. Moreover, it only takes 30 seconds to produce all 10 samples. While rcd improves the expressiveness over existing probabilistic constructs, it can be still evaluated in a accurate and efficient manner.

6.2.2 Inferring Robust Classifiers

In this case study, we show how to improve the robustness of a classifier using Omega. Machine learning models such as neural networks have been shown being vulnerable to adversarial inputs [**?**]: a small perturbation to the input can completely change the output. For security concerns, it is desirable that a machine learning model is resilient to such perturbations, in other words, robust. We do not claim that we have solved the problem nor that we intend to compete with existing techniques due to limitations in the underlying solvers. Instead, we use this use case to demonstrate the expressiveness of our language. Concretely, we will improve the robustness of a small neural network against a specific adversarial attack.

Setup. The neural network has three layers and two hidden neurons. Similar to the previous case study, it is also studied in [**?**] ($NN_{3,2}$ in Section 6) and trained on the same income dataset. We apply fast gradient sign method [**?**] to generate adversarial inputs. Out of 1,000 inputs that are randomly drawn from the input distribution, the network is resilient to the attack on 97% of the inputs. Our goal is to improve this ratio above 99%. We do not intend to make the network robust on arbitrary input because when the input space is continuous, inevitably there will be inputs that are very close to the decision boundary.

Probabilistic Program. Figure 7 outlines the Omega program we constructed for inferring parameters which make

the neural network robust. It begins with constructing random variables which represent the input distribution by invoking popModel(). Similar to the previous case study, popModel is implemented using a Bayesian network described in [**?**]. Then we construct distributions of neural network parameters, each of which is a normal whose mean is the corresponding original non-random parameter that is obtained by training on the dataset. Next, we construct perturbations to the inputs by using fast gradient sign method [**?**]. These perturbations are random variables that depend on the inputs and the parameters. Then we construct the robustness specification, which states that the output to the adversarial input should be the same as the original output with a probability greater than 0.99. Finally, we draw parameter samples from the random parameters conditioning on the robustness specification.

Accuracy and efficiency. To evaluate whether a sampled parameter leads to a robust classifier, we drew 1,000 samples from the input distribution, and applied fast gradient sign method to see if the attack changed the outputs. We drew 10 parameters and observed the resulted networks to be robust on 96.6%-100% inputs, with 98.8% being the average. Due to the approximation in evaluating prob and the underlying solver, our approach does not guarantee that the robustness specification is always satisfied. However, it does improve the robustness of the network for most cases. Moreover, these 10 parameters are produced in only 86 seconds.

7 Related Work

This contribution builds upon a long history of incorporating probability into programming languages [**? ? ? ?**]. Most probabilistic programming languages define samplers: procedures which invoke a random source of entropy [**?**]. In contrast, in Omega there is a strict separation between modeling and sampling. Defining a probabilistic program means to construct a collection of random variables on a shared probability space. A variety of both old and recent work [**? ?**] has gone into defining measure-theoretic based semantics for sampling based probabilistic programming languages. Omega instead has measure-theoretic objects as its primitive constructs.

For particular cases, *nested inference* has been used to achieve the same effects as random conditional distributions. They emerged out of the need to express nested goals [**?**] for ProbLog [**?**] and to model recursive reasoning [**?**]. The work of [**?**] aimed to mimic the capability of Prolog in handling nested goals: a outer goal could use the inner goal's success probability. To this end, they allow calls to problog inference engine within the inside a model. In a similar way, [**?**] added nested capabilities to Church by first defining query as a Church function. More recently, the programming language WebPPL [**?**] supports inference inside a model, which allows to compute the Kullback-Leibler divergence between two random distributions for optimal experiment design [**?**]. A review on *nested probabilistic programs* with focus on inference can be found in [**?**]

Algorithmic fairness and robustness have received much attention recently. In machine learning, various fairness specifications [**? ? ? ?**] have been proposed and many methods [**? ? ? ? ?**] have been designed to train a model to satisfy these specifications. These methods typically formulate the problem of training a fair classifier as a constrained or unconstrained optimization problem over the training data set to balance the tradeoff between accuracy and fairness. Similarly, many techniques have been proposed to improve robustness of machine learning models, in particular, neural networks. These techniques include obfuscating gradients [**?**], adversarial training [**? ?**], cascade adversarial training [**?**], constrained optimization [**?**], and many others. We do not intend to compete against these methods for improving fairness and robustness, but instead use these two case studies to demonstrate the expressiveness of our approach. However, our approach does have an edge over existing approaches when the user has prior knowledge about the input distribution.

In some cases it is possible to manually encode declarative knowledge constructively into a generative model. For example, one may specify a truncated normal distribution by conditioning a normal distribution to be bounded, or constructively using the inverse transform method. However, in most cases there is no straightforward means. Continuing the glucose example, we may attempt to tie parameters by drawing the parameters for each patient from a shared stochastic process. However, this increases the complexity of the generative model and requires significant expertise to ensure that the new generative model indeed captures the high-level constraint without unnecessarily constraining the model in unwanted ways.