Towards Verified Stochastic Variational Inference for Probabilistic Programs

Wonyeol Lee, Hangyeol Yu, Xavier Rival, Hongseok Yang

Towards Verified Stochastic Variational Inference for Probabilistic Programs

WONYEOL LEE, School of Computing, KAIST, South Korea
HANGYEOL YU, School of Computing, KAIST, South Korea
XAVIER RIVAL, INRIA Paris, CNRS/École Normale Supérieure/PSL University, France
HONGSEOK YANG, School of Computing, KAIST, South Korea

Probabilistic programming is the idea of writing models from statistics and machine learning using program notations and reasoning about these models using generic inference engines. Recently its combination with deep learning has been explored intensely, which led to the development of so called deep probabilistic programming languages, such as Pyro, Edward and ProbTorch. At the core of this development lie inference engines based on stochastic variational inference algorithms. When asked to find information about the posterior distribution of a model written in such a language, these algorithms convert this posterior-inference query into an optimisation problem and solve it approximately by a form of gradient ascent or descent. In this paper, we analyse one of the most fundamental and versatile variational inference algorithms, called score estimator or REINFORCE, using tools from denotational semantics and program analysis. We formally express what this algorithm does on models denoted by programs, and expose implicit assumptions made by the algorithm on the models. The violation of these assumptions may lead to an undefined optimisation objective or the loss of convergence guarantee of the optimisation process. We then describe rules for proving these assumptions, which can be automated by static program analyses. Some of our rules use nontrivial facts from continuous mathematics, and let us replace requirements about integrals in the assumptions, such as integrability of functions defined in terms of programs' denotations, by conditions involving differentiation or boundedness, which are much easier to prove automatically (and manually). Following our general methodology, we have developed a static program analysis for the Pyro programming language that aims at discharging the assumption about what we call model-guide support match. Our analysis is applied to the eight representative model-guide pairs from the Pyro webpage, which include sophisticated neural network models such as AIR. It finds a bug in one of these cases, reveals a non-standard use of an inference engine in another, and shows that the assumptions are met in the remaining six cases.

CCS Concepts: • **Mathematics of computing → Bayesian computation**; **Variational methods**; • **Theory of computation → Probabilistic computation**; **Denotational semantics**; • **Software and its engineering → Correctness**; **Automated static analysis**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: Probabilistic programming, static analysis, semantics, correctness

ACM Reference Format:
Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards Verified Stochastic Variational Inference for Probabilistic Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 16 (January 2020), 33 pages. https://doi.org/10.1145/3371084

Authors' addresses: Wonyeol Lee, School of Computing, KAIST, South Korea, wonyeol@kaist.ac.kr; Hangyeol Yu, School of Computing, KAIST, South Korea, yhk1344@kaist.ac.kr; Xavier Rival, INRIA Paris, CNRS/École Normale Supérieure/PSL University, France, rival@di.ens.fr; Hongseok Yang, School of Computing, KAIST, South Korea, hongseok.yang@kaist.ac.kr.

2475-1421/2020/1-ART16
https://doi.org/10.1145/3371084

1 INTRODUCTION

Probabilistic programming refers to the idea of writing models from statistics and machine learning using program notations and reasoning about these models using generic inference engines. It has been the subject of active research in machine learning and programming languages, because of its potential for enabling scientists and engineers to design and explore sophisticated models easily; when using these languages, they no longer have to worry about developing custom inference engines for their models, a highly-nontrivial task requiring expertise in statistics and machine learning. Several practical probabilistic programming languages now exist, and are used for a wide range of applications [Carpenter et al. 2017; Gehr et al. 2016; Goodman et al. 2008; Gordon et al. 2014; Mansinghka et al. 2014; Minka et al. 2014; Narayanan et al. 2016; Wood et al. 2014].

We consider inference engines that lie at the core of so called deep probabilistic programming languages, such as Pyro [Bingham et al. 2019], Edward [Tran et al. 2018, 2016] and ProbTorch [Siddharth et al. 2017]. These languages let users freely mix deep neural networks with constructs from probabilistic programming, in particular, those for writing Bayesian probabilistic models. In so doing, they facilitate the development of probabilistic deep-network models that may address the problem of measuring the uncertainty in current non-Bayesian deep-network models; a non-Bayesian model may predict that the price of energy goes up and that of a house goes down, but it cannot express, for instance, that the model is confident with the first prediction but not the second.

The primary inference engines for these deep probabilistic programming languages implement stochastic (or black-box) variational inference[1] algorithms. Converting inference problems into optimisation problems is the high-level idea of these algorithms.[2] When asked to find information about the posterior distribution of a model written in such a language, these algorithms convert this question to an optimisation problem and solve the problem approximately by performing a gradient descent or ascent on the optimisation objective. The algorithms work smoothly with gradient-based parameter-learning algorithms for deep neural networks, which is why they form the backbone for deep probabilistic programming languages.

In this paper, we analyse one of the most fundamental and versatile variational inference algorithms, called score estimator or REINFORCE[3] [Paisley et al. 2012; Ranganath et al. 2014; Williams 1992; Wingate and Weber 2013], using tools from denotational semantics and program analysis [Cousot and Cousot 1977, 1979, 1992]. We formally express what this algorithm does on models denoted by probabilistic programs, and expose implicit assumptions made by the algorithm on the models. The violation of these assumptions can lead to undefined optimisation objective or the loss of convergence guarantee of the optimisation process. We then describe rules for proving these assumptions, which can be automated by static program analyses. Some of our rules use nontrivial facts from continuous mathematics, and let us replace requirements about integrals in the assumptions, such as integrability of functions defined in terms of programs' denotations, by the conditions involving differentiation or boundedness, which are much easier to prove automatically (and manually) than the original requirements.

Following our general methodology, we have developed a static program analysis for the Pyro programming language that can discharge one assumption of the inference algorithm about so

[1] The term *stochastic* variational inference (VI) often refers to VI with data subsampling [Hoffman et al. 2013], and our usage of the term is often called black-box VI [Ranganath et al. 2014] to stress the treatment of a model as a black-box sampler.
[2] The inference problems in their original forms involve solving summation/integration/counting problems, which are typically more difficult than optimisation problems. The variational-inference algorithms convert the former problems to the latter ones, by looking for approximate, not exact, answers to the former.
[3] REINFORCE [Williams 1992] is an algorithm originally developed for reinforcement learning (RL), but it is commonly used as a synonym of the score-estimator algorithm. This is because REINFORCE and score estimator use a nearly identical method for estimating the gradient of an optimisation objective.

called model-guide pairs. In Pyro and other deep probabilistic programming languages, a program denoting a model typically comes with a companion program, called guide, decoder, or inference network. This companion, which we call guide, helps the inference algorithm to find a good approximation to what the model ultimately denotes under a given dataset (i.e., the posterior distribution of the model under the dataset); the algorithm uses the guide to fix the search space of approximations, and solves an optimisation problem defined on that space. A model and a guide should satisfy an important correspondence property, which says that they should use the same sets of random variables, and for any such random variable, if the probability of the variable having a particular value is zero in the model, it should also be zero in the guide. If the property is violated, the inference algorithm may attempt to solve an optimisation problem with undefined optimisation objective and return parameter values that do not make any sense. Our static analysis checks this correspondence property for Pyro programs. When applied to eight representative model-guide pairs from the Pyro webpage, which include sophisticated neural network models such as Attend-Infer-Repeat (AIR), the analysis found a bug in one of these cases, revealed a non-standard use of the inference algorithm in another, and proved that the property holds in the remaining six cases.

Another motivation for this paper is to demonstrate an opportunity for programming languages and verification research to have an impact on the advances of machine learning and AI technologies. One popular question is: what properties should we verify on machine-learning programs? Multiple answers have been proposed, which led to excellent research results, such as those on robustness of neural networks [Mirman et al. 2018]. But most of the existing research focuses on the final outcome of machine learning algorithms, not the process of applying these algorithms. One of our main objectives is to show that the process often relies on multiple assumptions on models and finding automatic ways for discharging these assumptions can be another way of making PL and verification techniques contribute. While our suggested solutions are not complete, they are intended to show the richness of this type of problems in terms of theory and practice.

We summarise the contributions of the paper:

- We formally express the behaviour of the most fundamental variational inference algorithm on probabilistic programs using denotational semantics, and identify requirements on program denotations that are needed for this algorithm to work correctly.
- We describe conditions that imply the identified requirements but are easier to prove. The sufficiency of the conditions relies on nontrivial results from continuous mathematics. We sketch a recipe for building program analyses for checking these conditions automatically.
- We present a static analysis for the Pyro language that checks the correspondence requirement of model-guide pairs. The analysis is based on our recipe, but extends it significantly to address challenges for dealing with features of the real-world language. Our analysis has successfully verified 6 representative Pyro model-guide examples, and found a bug in one example.

The extended version of the paper can be found in [Lee et al. 2019].

2 VARIATIONAL INFERENCE AND VERIFICATION CHALLENGES BY EXAMPLES

We start by explaining informally the idea of stochastic variational inference (in short SVI), one fundamental SVI algorithm, and the verification challenges that arise when we use this algorithm.

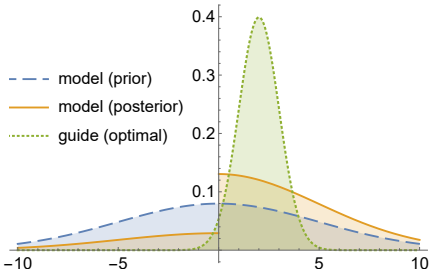
2.1 Stochastic Variational Inference

In a probabilistic programming language, we specify a model by a program. The program `model()` in Figure 1(a) is an example. It describes a joint probability density $p(v, obs)$ on two real-valued random variables v and obs. The value of the former is not observed, while the latter is observed to have the value 0. Finding out the value of v is the objective of writing this model. The joint

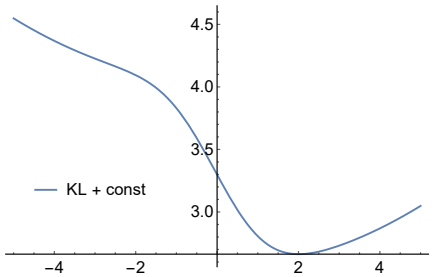
```
1  # define model and guide
2  def model():
3    v = pyro.sample("v", Normal(0., 5.))
4    if (v > 0):
5      pyro.sample("obs", Normal(1., 1.), obs=0.)
6    else:
7      pyro.sample("obs", Normal(-2., 1.), obs=0.)
8
9  def guide():
10   theta = pyro.param("theta", 3.)
11   v = pyro.sample("v", Normal(theta, 1.))
```

```
12 # perform stochastic variational inference
13 svi = SVI(model, guide,
14         Adam({"lr": 1.0e-2}),
15         loss=Trace_ELBO())
16 for step in range(2000):
17   svi.step()
18
19 # print result
20 print("trained theta =",
21   pyro.param("theta").item())
```

(a) Example model-guide pair for stochastic variational inference in Pyro.



(b) Probability densities of the model and the guide as a function of $v \in \mathbb{R}$.



(c) KL divergence from the guide to the model (plus $\log p(obs{=}0)$) as a function of $\theta \in \mathbb{R}$.

Fig. 1. Example of performing stochastic variational inference.

density $p(v, obs)$ is expressed in terms of *prior* $p(v)$ and *likelihood* $p(obs|v)$ in the program. The prior $p(v)$ of v is the normal distribution with mean 0 and standard deviation 5, and it expresses the belief about the possible value of v before any observation. The likelihood $p(obs|v)$ is a normal distribution whose mean and standard deviation are either $(1, 1)$ or $(-2, 1)$ depending on the sign of the value of v. The purpose of most inference algorithms is to compute exactly or approximately the *posterior* density given a prior and a likelihood. In our example, the posterior $p(v|obs{=}0)$ is:

$$p(v|obs{=}0) = \frac{p(v, obs{=}0)}{\int \mathrm{d}v\, p(v, obs{=}0)} = \frac{p(v) \cdot p(obs{=}0|v)}{p(obs{=}0)}.$$

Intuitively, the posterior expresses an updated belief on v upon observing $obs = 0$. The dashed blue and solid orange lines in Figure 1(b) show the prior and posterior densities, respectively. Note that the density of a positive v in the prior went up in the posterior. This is because when $v > 0$, the mean of $p(obs|v)$ is 1, a value closer to the observed value 0 than the alternative -2 for the mean.

SVI algorithms approach the posterior inference problem from the optimisation angle. They consider a collection of approximating distributions to a target posterior, formulate the problem of finding a good approximation in the collection as an optimisation problem, and solve the optimisation problem. The solution becomes the result of those algorithms. In Pyro, the users specify such a collection by a single parameterised program called *guide*; the collection can be generated by instantiating the parameters with different values. The program `guide()` in Figure 1(a) is an example. It has a real-valued parameter θ (written as `theta` in the program), and states that the probability density $q_\theta(v)$ of v is the normal distribution with unknown mean θ and standard deviation 1. The lines 13–17 in the figure show how to apply a standard SVI engine of Pyro (called `Trace_ELBO`) to find a good θ. They instruct the engine to solve the following optimisation problem:

$$\mathrm{argmin}_\theta \mathrm{KL}(q_\theta(v)||p(v|obs{=}0)), \qquad \text{where } \mathrm{KL}(q_\theta(v)||p(v|obs{=}0)) \triangleq \mathbb{E}_{q_\theta(v)}\left[\log \frac{q_\theta(v)}{p(v|obs{=}0)}\right].$$

The optimisation objective $\mathrm{KL}(q_\theta(v)||p(v|obs{=}0))$ is the *KL divergence* from $q_\theta(v)$ to $p(v|obs{=}0)$, and measures the similarity between the two densities, having a small value when the densities are similar. The KL divergence is drawn in Figure 1(c) as a function of θ, and the dotted green line in Figure 1(b) draws the density q_θ at the optimum θ. Note that the mean of this distribution is biased toward the positive side, which reflects the fact that the property $v > 0$ has a higher probability than its negation $v \leq 0$ in the posterior distribution.

One of the most fundamental and versatile algorithms for SVI is score estimator (also called REINFORCE). It repeatedly improves θ in two steps. First, it estimates the gradient of the optimisation objective with samples from the current q_{θ_n}:

$$\nabla_\theta \mathrm{KL}(q_\theta(v)||p(v|obs{=}0))\Big|_{\theta=\theta_n} \approx \frac{1}{N}\sum_{i=1}^{N}(\nabla_\theta \log q_{\theta_n}(v_i)) \cdot \log \frac{q_{\theta_n}(v_i)}{p(v_i, obs{=}0)}$$

where $v_1, \ldots, v_N$ are independent samples from the distribution q_{θ_n}. Then, the algorithm updates θ with the estimated gradient (the specific learning rate 0.01 is chosen to improve readability):

$$\theta_{n+1} \leftarrow \theta_n - 0.01 \times \frac{1}{N}\sum_{i=1}^{N}(\nabla_\theta \log q_{\theta_n}(v_i)) \cdot \log \frac{q_{\theta_n}(v_i)}{p(v_i, obs{=}0)}.$$

When the learning rate 0.01 is adjusted according to a known scheme, the algorithm is guaranteed to converge to a local optimum (in many cases) because its gradient estimate satisfies the following unbiasedness property (in those cases):

$$\nabla_\theta \mathrm{KL}(q_\theta(v)||p(v|obs{=}0))\Big|_{\theta=\theta_n} = \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N}(\nabla_\theta \log q_{\theta_n}(v_i)) \cdot \log \frac{q_{\theta_n}(v_i)}{p(v_i, obs{=}0)}\right] \tag{1}$$

where the expectation is taken over the independent samples $v_1, \ldots, v_N$ from q_{θ_n}.

2.2 Verification Challenges

We now give two example model-guide pairs that illustrate verification challenges related to SVI.

The first example appears in Figure 2(a). It is the Bayesian regression example from the Pyro webpage (this example is among the benchmarks used in §8), which solves the problem of finding a line that interpolates a given set of points in $\mathbb{R}^2$.

The problem with this example is that the KL divergence of its model-guide pair, the main optimisation objective in SVI, is undefined. The model and guide in the figure use the random variable `sigma`, but they use different non-zero-probability regions, called supports, for it. In the model, the support is $[0, 10]$, while that in the guide is $\mathbb{R}$. But the KL divergence from a guide to a model is defined only if for every random variable, its support in the guide is included in that in the model. We point out that this support mismatch was found by our static analyser explained in §8.

Figures 2(b) and 2(c) show two attempts to resolve the undefined-KL issue. To fix the issue, we change the distribution of `sigma` in the model in (b), and in the guide in (c). These revisions remove the problem about the support of `sigma`, but do not eliminate that of the undefined KL. In both (b) and (c), the KL divergence is ∞. This happens mainly because `sigma` can be arbitrarily close to 0 in the guide in both cases, which makes integrand in the definition of the KL divergence diverge to ∞.

An SVI-specific verification challenge related to this example is how to prove the well-definedness of the KL divergence and more generally the optimisation objective of an SVI algorithm. In §6.2, we provide a partial answer to the question. We give a condition for ensuring the well-definedness of the KL divergence. Our condition is more automation-friendly than the definition of KL, because it does not impose the difficult-to-check integrability requirement present in the definition of KL.

```
def model(...):
  ...
  sigma = pyro.sample("sigma",
            Uniform(0., 10.))
  ...
  pyro.sample("obs",
    Normal(..., sigma), obs=...)
```

```
def guide(...):
  ...
  loc = pyro.param("sigma_loc", 1.,
          constraint=constraints.positive)
  ...
  sigma = pyro.sample("sigma",
            Normal(loc, 0.05))
```

(a) Bayesian regression example from the Pyro webpage.

```
def model_r1(...):
  ...
  sigma = pyro.sample("sigma",
            Normal(5., 5.))
  ...
  pyro.sample("obs",
    Normal(..., abs(sigma)), obs=...)

def guide_r1(...):
  # same as guide() in (a)
  ...
```

(b) The example with a revised model.

```
def model_r2(...):
  # same as model() in (a)
  ...

def guide_r2(...):
  ...
  sigma = pyro.sample("sigma",
            Uniform(0., 10.))
```

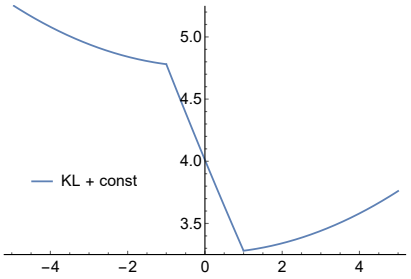
(c) The example with a revised guide.

Fig. 2. Example model-guide pairs whose KL divergence is undefined.

```
def model():
  v = pyro.sample("v", Normal(0., 5.))
  if (v > 0):
    pyro.sample("obs", Normal(1., 1.), obs=0.)
  else:
    pyro.sample("obs", Normal(-2., 1.), obs=0.)

def guide():
  theta = pyro.param("theta", 3.)
  v = pyro.sample("v",
        Uniform(theta-1., theta+1.))
```

(a) The model from Figure 1(a), and a guide using a parameterised uniform distribution.



(b) KL divergence from the guide to the model (plus $\log p(obs{=}0)$) as a function of $\theta \in \mathbb{R}$.

Fig. 3. Example model-guide pair for which the gradient of the KL divergence is undefined, or the score estimator is biased.

The second example appears in Figure 3(a). It uses the same model as in Figure 1(a), but has a new guide that uses a uniform distribution parameterised by $\theta \in \mathbb{R}$. For this model-guide pair, the KL divergence is well-defined for all $\theta \in \mathbb{R}$, and the optimal θ^* minimising the KL is $\theta^* = 1$.

However, as shown in Figure 3(b), the gradient of the KL divergence is undefined for $\theta \in \{-1, 1\}$, because the KL divergence is not differentiable at -1 and 1. For all the other $\theta \in \mathbb{R} \setminus \{-1, 1\}$, the KL divergence and its gradient are both defined, but the score estimator cannot estimate this gradient in an unbiased manner (i.e., in a way satisfying (1)), thereby losing the convergence guarantee to a local optimum. The precise calculation is not appropriate in this section, but we just point out that the expectation of the estimated gradient is always zero for all $\theta \in \mathbb{R} \setminus \{-1, 1\}$, but the true gradient of the KL is always non-zero for those θ, because it has the form: $\frac{\theta}{25} - \mathbb{1}_{[-1 \le \theta \le 1]} \cdot \frac{1}{2} \log \frac{\mathcal{N}(0;1,1)}{\mathcal{N}(0;-2,1)}$. Here $\mathcal{N}(v; \mu, \sigma)$ is the density of the normal distribution with mean μ and standard deviation σ (concretely, $\mathcal{N}(v; \mu, \sigma) = 1/(\sqrt{2\pi}\sigma) \cdot \exp(-(v-\mu)^2/(2\sigma^2))$). The mismatch comes from the invalidity of one implicit assumption about interchanging integration and gradient in the justification of the score estimator; see §5 for detail.

To sum up, the second example shows that even if the KL divergence is defined, its gradient is sometimes undefined, and also that even if both the KL divergence and its gradient are defined, the sample-based estimate of the gradient in a standard SVI algorithm may be biased—this means that the equation similar to (1) does not hold and an SVI algorithm is no longer guaranteed to converge to a local optimum. Proving that these failure cases do not arise is another SVI-specific verification challenge. In §6.3, we give another example of similar flavour, and provide an automation-friendly condition that ensures the existence of the KL divergence and its gradient as well as the unbiasedness of the gradient estimate of the score estimator.

We conclude the section by emphasising that the aforementioned issues could have a large impact on the results of SVI. For instance, running the Bayesian regression example in Figure 2(a) in Pyro after a small change (0.4 instead of 0.05 in line 14) sometimes results in a crash. Also, running its revised version in Figure 2(c) leads to the complete failure of an optimiser and produces meaningless results. These observations strengthen the importance of resolving the verification challenges presented above.[4]

3 REVIEW OF MEASURE THEORY AND NOTATIONS

A *σ-algebra* Σ on a set X is a collection of subsets of X such that (i) $X \in \Sigma$; (ii) $A_0 \cup A_1 \in \Sigma$ and $X \setminus A_0 \in \Sigma$ for all $A_0, A_1 \in \Sigma$; (iii) $\bigcup_n A_n \in \Sigma$ when all subsets A_n are in Σ. An equivalent but easier-to-remember characterisation is that Σ is closed under boolean operations and countable union and intersection. We call the pair (X, Σ) of a set and a σ-algebra *measurable space*, and subsets in Σ *measurable*. A function f from a measurable space (X, Σ) to another measurable space (X', Σ') is *measurable* if $f^{-1}(B) \in \Sigma$ for all $B \in \Sigma'$.

An example of measurable space is the n-dimensional Euclidean space $\mathbb{R}^n$ with the *Borel σ-algebra* $\mathcal{B} \triangleq \sigma(\{(-\infty, r_1) \times \cdots \times (-\infty, r_n) \mid r \in \mathbb{R}^n\})$, where σ is the closure operator that converts a collection of subsets of $\mathbb{R}^n$ into the smallest σ-algebra containing the collection. Subsets X of $\mathbb{R}^n$, such as $[0, \infty)^n$, form measurable spaces with the σ-algebra $\{X \cap A \mid A \in \mathcal{B}\}$. Another example is a set X with the so called *discrete* σ-algebra on X that consists of all subsets of X.

A *measure* μ on a measurable space (X, Σ) is a function from Σ to $[0, \infty]$ such that $\mu(\emptyset) = 0$ and μ satisfies the *countable additivity condition*: $\mu(\bigcup_{n=0}^{\infty} B_n) = \sum_{n=0}^{\infty} \mu(B_n)$ for any *countable* family of *disjoint* measurable subsets B_n. A well-known example is the Lebesgue measure λ^n on $\mathbb{R}^n$ which maps each measurable subset of $\mathbb{R}^n$ to its volume in the usual sense.[5] When $\mu(X) \leq 1$, we call μ *subprobability measure*. If $\mu(X) = 1$, we may drop "sub", and call μ *probability measure*.

The Lebesgue integral $\int$ is a partial operator that maps a measure μ on (X, Σ) and a real-valued measurable function on the same space (X, Σ) to a real number. It is denoted by $\int \mu(\mathrm{d}x)\, f(x)$. To follow the paper, it is enough to know that this integral generalises the usual Riemann integral from calculus.[6] For a measure ν on (X, Σ), if $\nu(A) = \int \mu(\mathrm{d}x)\, (f(x) \cdot \mathbb{1}_{[x \in A]})$ for non-negative f, we say that f is the *density* of ν with respect to μ and call μ *reference measure*.

In the paper, we use a few well-known methods for building measurable spaces.

The first method applies when we are given a set X and a collection of functions $\{f_i : X \to X_i \mid i \in I\}$ to measurable spaces (X_i, Σ_i). The method is to equip X with the smallest σ-algebra Σ making all f_i's measurable: $\Sigma \triangleq \sigma(\{f_i^{-1}(B) \mid i \in I,\ B \in \Sigma_i\})$.

[4] The aforementioned issues do not always cause problems in inference results (e.g., Figures 2(a) and 2(c) mostly give reasonable results), because the random seeds and the initial values for SVI could be set well so that the probability of SVI going wrong becomes low. We emphasise, however, that the probability is non-zero.

[5] The Lebesgue measure λ^n is the unique measure on $\mathbb{R}^n$ that sets the volume of the unit cube $(0, 1)^n$ to 1 and is translation invariant: for all measurable subsets A and $r \in \mathbb{R}^n$, $\lambda^n(A) = \lambda^n(\{r' - r \mid r' \in A\})$.

[6] Another useful fact is that when f is non-negative, $\int \mu(\mathrm{d}x) f(x) = \sup \sum_i (\inf_{x \in A_i} f(x)) \cdot \mu(A_i)$ where the supremum is taken with respect to all finite partitions $\{A_i\}_i$ of X into measurable subsets.

The second relies on two constructions for sets, i.e., product and disjoint union. Suppose that we are given measurable spaces (X_i, Σ_i) for all $i \in I$. We define a *product measurable space* that has $\prod_{i\in I} X_i$ as its underlying set and the following product σ-algebra $\bigotimes_{i\in I} \Sigma_i$ as its σ-algebra:

$$\bigotimes_{i\in I} \Sigma_i \triangleq \sigma\Big(\Big\{ \prod_i A_i \;\Big|\; \text{there is a finite } I_0 \subseteq I \text{ such that } (\forall j \in I \setminus I_0.\, A_j = X_j) \wedge (\forall i \in I_0.\, A_i \in \Sigma_i) \Big\}\Big).$$

The construction of the product σ-algebra can be viewed as a special case of the first method where we consider the smallest σ-algebra on $\prod_{i\in I} X_i$ that makes every projection map to X_i measurable. When the X_i are disjoint, they can be combined as disjoint union. The underlying set in this case is $\bigcup_{i\in I} X_i$, and the σ-algebra is

$$\bigoplus_{i\in I} \Sigma_i \triangleq \{A \mid A \cap X_i \in \Sigma_i \text{ for all } i \in I\}.$$

When $I = \{i, j\}$ with $i \neq j$, we denote the product measurable space by $(X_i \times X_j, \Sigma_i \otimes \Sigma_j)$. In addition, if X_i and X_j are disjoint, we write $(X_i \cup X_j, \Sigma_i \oplus \Sigma_j)$ for the disjoint-union measurable space.

The third method builds a measurable space out of measures or a certain type of measures, such as subprobability measures. For a measurable space (X, Σ), we form a measurable space with measures. The underlying set $\mathrm{Mea}(X)$ and σ-algebra $\Sigma_{\mathrm{Mea}(X)}$ of the space are defined by

$$\mathrm{Mea}(X) \triangleq \{\mu \mid \mu \text{ is a measure on } (X, \Sigma)\}, \qquad \Sigma_{\mathrm{Mea}(X)} \triangleq \sigma\big(\big\{\{\mu \mid \mu(A) \leq r\} \;\big|\; A \in \Sigma, r \in \mathbb{R}\big\}\big).$$

The difficult part to grasp is $\Sigma_{\mathrm{Mea}(X)}$. Once again, a good approach for understanding it is to realise that $\Sigma_{\mathrm{Mea}(X)}$ is the smallest σ-algebra that makes the function $\mu \longmapsto \mu(A)$ from $\mathrm{Mea}(X)$ to $\mathbb{R}$ measurable for all measurable subsets $A \in \Sigma$. This measurable space gives rise to a variety of measurable spaces, each having a subset M of $\mathrm{Mea}(X)$ as its underlying set and the induced σ-algebra $\Sigma_M = \{A \cap M \mid A \in \Sigma_{\mathrm{Mea}(X)}\}$. In the paper, we use two such spaces, one induced by the set $\mathrm{Sp}(X)$ of subprobability measures on X and the other by the set $\mathrm{Pr}(X)$ of probability measures.

A measurable function f from (X, Σ) to $(\mathrm{Mea}(Y), \Sigma_{\mathrm{Mea}(Y)})$ is called *kernel*. If $f(x)$ is a subprobability measure (i.e., $f(x) \in \mathrm{Sp}(Y)$) for all x, we say that f is a *subprobability kernel*. In addition, if $f(x)$ is a probability measure (i.e., $f(x) \in \mathrm{Pr}(Y)$) for all x, we call f *probability kernel*. A good heuristic is to view a probability kernel as a random function and a subprobability kernel as a random partial function. We use well-known facts that a function $f : X \to \mathrm{Mea}(Y)$ is a subprobability kernel if and only if it is a measurable map from (X, Σ) to $(\mathrm{Sp}(Y), \Sigma_{\mathrm{Sp}(Y)})$, and that similarly a function f is a probability kernel if and only if it is a measurable function from (X, Σ) to $(\mathrm{Pr}(Y), \Sigma_{\mathrm{Pr}(Y)})$.

We use a few popular operators for constructing measures throughout the paper. We say that a measure μ on a measurable space (X, Σ) is *finite* if $\mu(X) < \infty$, and *σ-finite* if there is a countable partition of X into measurable subsets X_n's such that $\mu(X_n) < \infty$ for every n. Given a finite or countable family of σ-finite measures $\{\mu_i\}_{i\in I}$ on measurable spaces (X_i, Σ_i)'s, the product measure of μ_i's, denoted $\bigotimes_{i\in I} \mu_i$, is the unique measure on $(\prod_{i\in I} X_i, \bigotimes_{i\in I} \Sigma_i)$ such that $(\bigotimes_{i\in I} \mu_i)(\prod_{i\in I} A_i) = \prod_{i\in I} \mu_i(A_i)$ for all measurable subsets A_i of X_i. Given a finite or countable family of measures $\{\mu_i\}_{i\in I}$ on disjoint measurable spaces (X_i, Σ_i)'s, the sum measure of μ_i's, denoted $\bigoplus_{i\in I} \mu_i$, is the unique measure on $(\sum_{i\in I} X_i, \bigoplus_{i\in I} \Sigma_i)$ such that $(\bigoplus_{i\in I} \mu_i)(\bigcup_{i\in I} A_i) = \sum_{i\in I} \mu_i(A_i)$.

Throughout the paper, we take the convention that the set $\mathbb{N}$ of natural numbers includes 0. For all positive integers n, we write $[n]$ to mean the set $\{1, 2, \ldots, n\}$.

4 SIMPLE PROBABILISTIC PROGRAMMING LANGUAGE

In this section, we describe the syntax and semantics of a simple probabilistic programming language, which we use to present the theoretical results of the paper. The measure semantics in

$$
\begin{array}{rrcl@{\qquad}rrcl}
\textit{Real constants} & c & \in & \mathbb{R} & \textit{Primitive real-valued functions} & f & ::= & \ldots \\
\textit{String constants} & \alpha & \in & \textit{Str} & \textit{Primitive string-valued functions} & g & ::= & \ldots \\
\textit{Real expressions} & E & ::= & c \mid x \mid f(E, \ldots, E) & & & & \\
\textit{Boolean expressions} & B & ::= & \mathsf{true} \mid E < E \mid B \wedge B \mid \neg B & & & & \\
\textit{String expressions} & S & ::= & \alpha \mid g(S, \ldots, S, E, \ldots, E) & & & & \\
\textit{Commands} & C & ::= & \mathsf{skip} \mid x := E \mid C; C \mid \mathsf{if}\, B\, \{C\}\, \mathsf{else}\, \{C\} \mid \mathsf{while}\, B\, \{C\} & & & & \\
 & & & \mid x := \mathsf{sample}_{\mathsf{norm}}(S, E, E) \mid \mathsf{score}_{\mathsf{norm}}(E, E, E) & & & &
\end{array}
$$

Fig. 4. Grammar of our core language.

§4.2 uses results and observations from [Staton et al. 2016], but the density semantics and the other materials in §4.3 are new in this work.

4.1 Syntax

We use an extension of the standard while language with primitives for probabilistic programming. The grammar of the language is given in Figure 4. Variables in the language store real numbers, but expressions may denote reals, booleans and strings, and they are classified into E, B, S based on these denoted values. The primitive functions f for reals and g for strings may be usual arithmetic and string operations, such as multiplication and exponentiation for f and string concatenation for g.

The grammar for C includes the cases for the standard constructs of the while language, such as assignment, sequencing, conditional statement and loops. In addition, it has two constructs for probabilistic programming. The first $x := \mathsf{sample}_{\mathsf{norm}}(S, E_1, E_2)$ draws a sample from the *normal* distribution with mean E_1 and standard deviation E_2 and names the sample with the string S. The next $\mathsf{score}_{\mathsf{norm}}(E_0, E_1, E_2)$ expresses that a sample is drawn from the normal distribution with mean E_1 and standard deviation E_2 and the value of this sample is observed to be E_0. It lets the programmers express information about observed data inside programs. Operationally, this construct can be understood as an instruction for updating a global variable that stores the so called importance score of the execution. The importance score quantitatively records how well the random choices in the current execution match the observations, and the score statement updates this score by multiplying it with the density at E_0 of the appropriate normal distribution.

Consider the following program:

$$x := \mathsf{sample}_{\mathsf{norm}}(\text{“}a\text{”}, 0.0, 5.0);\ \mathsf{score}_{\mathsf{norm}}(3.0, x, 1.0).$$

The program specifies a model with one random variable “a”. Using a relatively flat normal distribution, the program specifies a prior belief that the value of the random variable “a” is likely to be close to 0.0 and lie between -2×5.0 and 2×5.0. The next score statement refines this belief with one data point 3.0, which is a noisy observation of the value of “a” (bound to x). The parameters to the normal density in the statement express that the noise is relatively small, between -2×1.0 and 2×1.0. Getting the refined belief, called posterior distribution, is the reason that a data scientist writes a model like this program. It is done by an inference algorithm of the language.

Permitting only the normal distribution does not limit the type of models expressible in the language. Every distribution can be obtained by transforming the standard normal distribution.[7]

[7]Here we consider only Borel spaces. Although using only the normal distribution does not affect the expressiveness, it has an impact on stochastic variational inference to be discussed later, because it requires a guide to use only normal distributions. But the impact is not significant, because most well-known approaches for creating guides from the machine-learning literature (such as extensions of variational autoencoder) use normal distributions only, or can be made to do so easily.

4.2 Measure Semantics

The denotational semantics of the language just presented is mostly standard, but employs some twists to address the features for probabilistic programming [Staton et al. 2016].

Here is a short high-level overview of the measure semantics. Our semantics defines multiple measurable spaces, such as *Store* and *State*, that hold mathematical counterparts to the usual actors in computation, such as program stores (i.e., mappings from variables to values) and states (which consist of a store and further components). Then, the semantics interprets expressions E, B, S and commands C as measurable functions of the following types:

$$[\![E]\!] : \mathit{Store} \to \mathbb{R}, \quad [\![B]\!] : \mathit{Store} \to \mathbb{B}, \quad [\![S]\!] : \mathit{Store} \to \mathit{Str}, \quad [\![C]\!] : \mathit{State} \to \mathrm{Sp}(\mathit{State} \times [0, \infty)).$$

Here $\mathbb{R}$ is the measurable space of reals with the Borel σ-algebra, and $\mathbb{B}$ and *Str* are discrete measurable spaces of booleans and strings. *Store* and *State* are measurable spaces for stores (i.e., maps from variables to values) and states which consist of a store and a part for recording information about sampled random variables. Note that the target measurable space of commands is built by first taking the product of measurable spaces *State* and $[0, \infty)$ and then forming a space out of subprobability measures on $\mathit{State} \times [0, \infty)$. This construction indicates that commands denote probabilistic computations, and the result of each such computation consists of an output state and a score which expresses how well the computation matches observations expressed with the score statements in the command C. Some of the possible outcomes of the computation may lead to non-termination or an error, and these abnormal outcomes are not accounted for by the semantics, which is why $[\![C]\!](\sigma)$ for a state $\sigma \in \Sigma$ is a subprobability distribution. The semantics of expressions is much simpler. It just says that expressions do not involve any probabilistic computations, so that they denote deterministic measurable functions.

We now explain how this high-level idea gets implemented in our semantics. Let *Var* be a countably infinite set of variables. Our semantics uses the following sets:

$$\begin{aligned}
\textit{Stores} \quad & s \in \mathit{Store} \triangleq [\mathit{Var} \to \mathbb{R}] \quad \Big(\text{which is isomorphic to } \prod_{x \in \mathit{Var}} \mathbb{R}\Big) \\
\textit{Random databases} \quad & r \in \mathit{RDB} \triangleq \bigcup_{K \subseteq_{\mathit{fin}} \mathit{Str}} [K \to \mathbb{R}] \quad \Big(\text{which is isomorphic to } \bigcup_{K \subseteq_{\mathit{fin}} \mathit{Str}} \prod_{\alpha \in K} \mathbb{R}\Big) \\
\textit{States} \quad & \sigma \in \mathit{State} \triangleq \mathit{Store} \times \mathit{RDB},
\end{aligned}$$

where $[X \to Y]$ denotes the set of all functions from X to Y. A state σ consists of a store s and a random database r. The former fixes the values of variables, and the latter records the name (given as a string) and the value of each sampled random variable. The domain of r is the names of all the sampled random variables. By insisting that r should be a map, the semantics asserts that no two random variables have the same name. For each state σ, we write σ_s and σ_r for its store and random database components, respectively. Also, for a variable x, a string α and a value v, we write $\sigma[x \mapsto v]$ and $\sigma[\alpha \mapsto v]$ to mean $(\sigma_s[x \mapsto v], \sigma_r)$ and $(\sigma_s, \sigma_r[\alpha \mapsto v])$.

We equip all of these sets with σ-algebras and turn them to measurable spaces in a standard way. Note that we constructed the sets from $\mathbb{R}$ by repeatedly applying the product and disjoint-union operators. We equip $\mathbb{R}$ with the usual Borel σ-algebra. Then, we parallel each usage of the product and the disjoint-union operators on sets with that of the corresponding operators on σ-algebras. This gives the σ-algebras for all the sets defined above. Although absent from the above definition, the measurable spaces $\mathbb{B}$ and *Str* equipped with discrete σ-algebras are also used in our semantics.

We interpret expressions E, B, S as measurable functions $[\![E]\!] : \mathit{Store} \to \mathbb{R}$, $[\![B]\!] : \mathit{Store} \to \mathbb{B}$, and $[\![S]\!] : \mathit{Store} \to \mathit{Str}$, under the assumption that the semantics of primitive real-valued f of

$$
\begin{aligned}
[\![\mathsf{skip}]\!](\sigma)(A) &\triangleq \mathbb{1}_{[(\sigma,1)\in A]} \qquad [\![x := E]\!](\sigma)(A) \triangleq \mathbb{1}_{[(\sigma[x\mapsto[\![E]\!]\sigma_s],1)\in A]}\\
[\![C_0; C_1]\!](\sigma)(A) &\triangleq \int [\![C_0]\!](\sigma)(\mathrm{d}(\sigma', w')) \int [\![C_1]\!](\sigma')(\mathrm{d}(\sigma'', w'')) \, \mathbb{1}_{[(\sigma'', w'\cdot w'')\in A]}\\
[\![\mathsf{if}\ B\ \{C_0\}\ \mathsf{else}\ \{C_1\}]\!](\sigma)(A) &\triangleq \mathbb{1}_{[[\![B]\!]\sigma_s = true]} \cdot [\![C_0]\!](\sigma)(A) + \mathbb{1}_{[[\![B]\!]\sigma_s \neq true]} \cdot [\![C_1]\!](\sigma)(A)\\
[\![\mathsf{while}\ B\ \{C\}]\!](\sigma)(A) &\triangleq (\mathit{fix}\, F)(\sigma)(A)\\
\big(\text{where } F(\kappa)(\sigma)(A) &\triangleq \mathbb{1}_{[[\![B]\!]\sigma_s \neq true]} \cdot \mathbb{1}_{[(\sigma,1)\in A]}\\
&\quad + \mathbb{1}_{[[\![B]\!]\sigma_s = true]} \cdot \int [\![C]\!](\sigma)(\mathrm{d}(\sigma', w')) \int \kappa(\sigma')(\mathrm{d}(\sigma'', w'')) \, \mathbb{1}_{[(\sigma'', w'\cdot w'')\in A]}\big)\\
[\![x := \mathsf{sample}_{\mathsf{norm}}(S, E_1, E_2)]\!](\sigma)(A) &\triangleq \mathbb{1}_{[[\![S]\!]\sigma_s \notin \mathrm{dom}(\sigma_r)]} \cdot \mathbb{1}_{[[\![E_2]\!]\sigma_s \in (0,\infty)]}\\
&\quad \cdot \int \mathrm{d}v \, \big(\mathcal{N}(v; [\![E_1]\!]\sigma_s, [\![E_2]\!]\sigma_s) \cdot \mathbb{1}_{[((\sigma_s[x\mapsto v], \sigma_r[[\![S]\!]\sigma_s \mapsto v]),1)\in A]}\big)\\
[\![\mathsf{score}_{\mathsf{norm}}(E_0, E_1, E_2)]\!](\sigma)(A) &\triangleq \mathbb{1}_{[[\![E_2]\!]\sigma_s \in (0,\infty)]} \cdot \mathbb{1}_{[(\sigma, \mathcal{N}([\![E_0]\!]\sigma_s; [\![E_1]\!]\sigma_s, [\![E_2]\!]\sigma_s))\in A]}
\end{aligned}
$$

Fig. 5. Measure semantics $[\![C]\!] \in \mathcal{K}$ of commands C. Here $\mathcal{N}(v; \mu, \sigma)$ is the density of the normal distribution with mean μ and standard deviation σ.

arity n and string-valued g of arity (m, l) are given by measurable functions $[\![f]\!] : \mathbb{R}^n \to \mathbb{R}$ and $[\![g]\!] : \mathit{Str}^m \times \mathbb{R}^l \to \mathit{Str}$. It is standard, and we describe it only for some sample cases of E and S:

$$
\begin{aligned}
&[\![x]\!]s \triangleq s(x), \qquad [\![f(E_0, \ldots, E_{n-1})]\!]s \triangleq [\![f]\!]([\![E_0]\!]s, \ldots, [\![E_{n-1}]\!]s),\\
&[\![\alpha]\!]s \triangleq \alpha, \qquad [\![g(S_0, \ldots, S_{m-1}, E_0, \ldots, E_{l-1})]\!]s \triangleq [\![g]\!]([\![S_0]\!]s, \ldots, [\![S_{m-1}]\!]s, [\![E_0]\!]s, \ldots, [\![E_{l-1}]\!]s).
\end{aligned}
$$

Lemma 4.1. *For all expressions E, B, and S, their semantics $[\![E]\!]$, $[\![B]\!]$ and $[\![S]\!]$ are measurable functions from Store to $\mathbb{R}$, $\mathbb{B}$ and Str, respectively.*

We interpret commands C as measurable functions from *State* to $\mathrm{Sp}(\mathit{State} \times [0, \infty))$, i.e., subprobability kernels from *State* to *State* $\times [0, \infty)$. Let $\mathcal{K}$ be the set of subprobability kernels from *State* to *State* $\times [0, \infty)$, and $\Sigma_{\mathit{State}\times[0,\infty)}$ be the σ-algebra of the product space *State* $\times [0, \infty)$. We equip $\mathcal{K}$ with the partial order $\sqsubseteq$: for all $\kappa, \kappa' \in \mathcal{K}$, $\kappa \sqsubseteq \kappa'$ if and only if $\kappa(\sigma)(A) \leq \kappa'(\sigma)(A)$ for all $\sigma \in \mathit{State}$ and $A \in \Sigma_{\mathit{State}\times[0,\infty)}$. The next lemma is a minor adaptation of a known result [Kozen 1981].

Lemma 4.2. *$(\mathcal{K}, \sqsubseteq)$ is an ω-complete partial order with the least element $\bot \triangleq (\lambda\sigma.\, \lambda A.\, 0)$.*

The measure semantics of a command C is defined in Figure 5. The interpretation of the loop is the least fixed point of the function F on the ω-complete partial order $\mathcal{K}$. The function F is continuous, so the least fixed point is obtained by the ω-limit of the sequence $\{F^n(\bot)\}_n$. In the definition of F, the argument κ is a sub-probability kernel, and it represents the computation after the first iteration of the loop. The semantics of the sample statement uses an indicator function to exclude erroneous executions where the argument S denotes a name already used by some previous random variable, or the standard deviation $[\![E_2]\!]\sigma_s$ is not positive. When this check passes, it distils A to a property on the value of x and computes the probability of the property using the normal distribution with mean $[\![E_1]\!]\sigma_s$ and standard deviation $[\![E_2]\!]\sigma_s$.

Theorem 4.3. *For every command C, its interpretation $[\![C]\!]$ is well-defined and belongs to $\mathcal{K}$.*

4.3 Posterior Inference and Density Semantics

We write a probabilistic program to answer queries about the model and data that it describes. Among such queries, posterior inference is one of the most important and popular. Let $\sigma_I = (s_I, [])$ be the initial state that consists of some fixed store and the empty random database. In our setting, posterior inference amounts to finding information about the following probability measure $\Pr(C, \cdot)$ for a command C. For a measurable $A \subseteq \mathit{RDB}$,

$$
\mathrm{Mea}(C, A) \triangleq \int [\![C]\!](\sigma_I)(\mathrm{d}(\sigma, w))(w \cdot \mathbb{1}_{[\sigma \in \mathit{Store}\times A]}), \quad Z_C \triangleq \mathrm{Mea}(C, \mathit{RDB}), \quad \Pr(C, A) \triangleq \frac{\mathrm{Mea}(C, A)}{Z_C}. \tag{2}
$$

The probability measure $\Pr(C, \cdot)$ is called the *posterior distribution of* C, and $\mathrm{Mea}(C, \cdot)$ the *unnormalised posterior distribution of* C (in $\mathrm{Mea}(C, \cdot)$ and $\Pr(C, \cdot)$, we elide the dependency on s_I to avoid clutter). Finding information about the former is the goal of most inference engines of existing probabilistic programming languages. Of course, $\Pr(C, \cdot)$ is not defined when the normalising constant Z_C is infinite or zero. The inference engines regard such a case as an error that a programmer should avoid, and consider only C without those errors.

Most algorithms for posterior inference use the density semantics of commands. They implicitly pick measures on some measurable spaces used in the semantics. These measures are called *reference measures*, and constructed out of Lebesgue and counting measures [Bhat et al. 2012, 2013; Hur et al. 2015]. Then, the algorithms interpret commands as density functions with respect to these measures. One outcome of this density semantics is that the unnormalised posterior distribution $\mathrm{Mea}(C, \cdot)$ of a command C has a measurable function $f : \mathit{RDB} \to [0, \infty)$ such that $\mathrm{Mea}(C, A) = \int \rho(\mathrm{d}r)\,(\mathbb{1}_{[r \in A]} \cdot f(r))$, where ρ is a reference measure on RDB. Function f is called *density* of $\mathrm{Mea}(C, \cdot)$ with respect to ρ.

In the rest of this subsection, we define the meanings of commands using density functions. To do this, we need to set up some preliminary definitions.

First, we look at a predicate and an operator for random databases, which are about the possibility and the very act of merging two databases. For $r, r' \in \mathit{RDB}$, define the predicate $r \# r'$ by:

$$r \# r' \iff \mathrm{dom}(r) \cap \mathrm{dom}(r') = \emptyset.$$

When $r \# r'$, let $r \uplus r'$ be the random database obtained by merging r and r':

$$\mathrm{dom}(r \uplus r') \triangleq \mathrm{dom}(r) \cup \mathrm{dom}(r'); \qquad (r \uplus r')(\alpha) \triangleq \text{if } \alpha \in \mathrm{dom}(r) \text{ then } r(\alpha) \text{ else } r'(\alpha).$$

Lemma 4.4. *For every measurable* $h : \mathit{RDB} \times \mathit{RDB} \times \mathit{RDB} \to \mathbb{R}$, *the function* $(r, r') \longmapsto \mathbb{1}_{[r \# r']} \times h(r, r', r \uplus r')$ *from* $\mathit{RDB} \times \mathit{RDB}$ *to* $\mathbb{R}$ *is measurable.*

Second, we define a reference measure ρ on RDB:

$$\rho(R) \triangleq \sum_{K \subseteq_{\mathit{fin}} \mathit{Str}} \Big(\bigotimes_{\alpha \in K} \lambda \Big)(R \cap [K \to \mathbb{R}]),$$

where λ is the Lebesgue measure on $\mathbb{R}$. As explained in the preliminary section, the symbol $\otimes$ here represents the operator for constructing a product measure. In particular, $\bigotimes_{\alpha \in K} \lambda$ refers to the product of the $|K|$ copies of the Lebesgue measure λ on $\mathbb{R}$. In the above definition, we view functions in $[K \to \mathbb{R}]$ as tuples with $|K|$ real components and measure sets of such functions using the product measure $\bigotimes_{\alpha \in K} \lambda$. When K is the empty set, $\bigotimes_{\alpha \in K} \lambda$ is the nullary-product measure on $\{[]\}$, which assigns 1 to $\{[]\}$ and 0 to the empty set.

The measure ρ computes the size of each measurable subset $R \subseteq \mathit{RDB}$ in three steps. It splits a given R into groups based on the domains of elements in R. Then, it computes the size of each group separately, using the product of the Lebesgue measure. Finally, it adds the computed sizes. The measure ρ is not finite, but it satisfies the *σ-finiteness* condition,[8] the next best property.

Third, we define a partially-ordered set $\mathcal{D}$ with certain measurable functions. We say that a function $g : \mathit{Store} \times \mathit{RDB} \to \{\bot\} \cup (\mathit{Store} \times \mathit{RDB} \times [0, \infty) \times [0, \infty))$ *uses random databases locally* or *is local* if for all $s, s' \in \mathit{Store}$, $r, r' \in \mathit{RDB}$, and $w', p' \in [0, \infty)$,

$$\begin{aligned} g(s, r) = (s', r', w', p') \implies & (\exists r''.\, r' \# r'' \wedge r = r' \uplus r'' \wedge g(s, r'') = (s', [], w', p')) \\ & \wedge (\forall r'''.\, r \# r''' \implies g(s, r \uplus r''') = (s', r' \uplus r''', w', p')). \end{aligned}$$

[8] The condition lets us use Fubini theorem when showing the well-formedness of the density semantics in this subsection and relating this semantics with the measure semantics in the previous subsection.

$$
\begin{aligned}
[\![\mathsf{skip}]\!]_d(s,r) &\triangleq (s,r,1,1) \qquad [\![x{:=}E]\!]_d(s,r) \triangleq (s[x \mapsto [\![E]\!]s], r, 1, 1)\\
[\![C_0; C_1]\!]_d(s,r) &\triangleq ([\![C_1]\!]_d^{\ddagger} \circ [\![C_0]\!]_d)(s,r)\\
[\![\mathsf{if}\ B\ \{C_0\}\ \mathsf{else}\ \{C_1\}]\!]_d(s,r) &\triangleq \text{if } ([\![B]\!]s = \mathit{true}) \text{ then } [\![C_0]\!]_d(s,r) \text{ else } [\![C_1]\!]_d(s,r)\\
[\![\mathsf{while}\ B\ \{C\}]\!]_d(s,r) &\triangleq (\mathit{fix}\, G)(s,r)\\
&\quad \text{(where } G(g)(s,r) = \text{if } ([\![B]\!]s \neq \mathit{true}) \text{ then } (s,r,1,1) \text{ else } (g^{\ddagger} \circ [\![C]\!]_d)(s,r))\\
[\![x{:=}\mathsf{sample}_{\mathsf{norm}}(S,E_1,E_2)]\!]_d(s,r) &\triangleq \text{if } ([\![S]\!]s \notin \mathrm{dom}(r) \vee [\![E_2]\!]s \notin (0,\infty)) \text{ then } \bot\\
&\quad \text{else } (s[x \mapsto r([\![S]\!]s)],\ r \setminus [\![S]\!]s,\ 1,\ \mathcal{N}(r([\![S]\!]s); [\![E_1]\!]s, [\![E_2]\!]s))\\
[\![\mathsf{score}_{\mathsf{norm}}(E_0,E_1,E_2)]\!]_d(s,r) &\triangleq \text{if } ([\![E_2]\!]s \notin (0,\infty)) \text{ then } \bot \text{ else } (s, r, \mathcal{N}([\![E_0]\!]s; [\![E_1]\!]s, [\![E_2]\!]s), 1)
\end{aligned}
$$

Fig. 6. Density semantics $[\![C]\!]_d \in \mathcal{D}$ of commands C

The condition describes the way that g uses a given random database r, which plays the role of a bank of random seeds (that g may partially consume as it needs random values). Some part r'' of r may be consumed by g, but the unconsumed part r' of r does not change and is returned in the output. Also, the behaviour of g does not depend on the unconsumed r'. We define the set $\mathcal{D}$ by

$$\mathcal{D} \triangleq \left\{ g : \mathit{Store} \times \mathit{RDB} \to \{\bot\} \cup (\mathit{Store} \times \mathit{RDB} \times [0,\infty) \times [0,\infty)) \;\middle|\; g \text{ is measurable and local} \right\}. \quad (3)$$

Here we view $\{\bot\}$ and $[0,\infty)$ as measurable spaces equipped with discrete and Borel σ-algebras. Also, we regard $\mathit{Store} \times \mathit{RDB}$ and $\{\bot\} \cup (\mathit{Store} \times \mathit{RDB} \times [0,\infty) \times [0,\infty))$ as measurable spaces constructed by the product and disjoint-union operators on measurable spaces, as explained in §3.

The locality in the definition of $\mathcal{D}$ formalises expected behaviours of commands. In fact, as we will show shortly, it is satisfied by all commands in our density semantics. This property plays an important role when we establish the connection between the density semantics in this subsection and the standard measure semantics in §4.2.

The functions in $\mathcal{D}$ are ordered pointwise: for all $g, g' \in \mathcal{D}$,

$$g \sqsubseteq g' \iff \forall (s,r) \in \mathit{Store} \times \mathit{RDB}.\, (g(s,r) = \bot \vee g(s,r) = g'(s,r)).$$

Lemma 4.5. *$(\mathcal{D}, \sqsubseteq)$ is an ω-complete partial order and has the least element $a \longmapsto \bot$. Thus, every continuous function G on $\mathcal{D}$ has a least fixed point (and this least fixed point is unique).*

For each $g \in \mathcal{D}$, let $g^{\ddagger}$ be the following lifting to a function on $\{\bot\} \cup (\mathit{Store} \times \mathit{RDB} \times [0,\infty) \times [0,\infty))$:

$$g^{\ddagger}(\bot) \triangleq \bot, \qquad g^{\ddagger}(s,r,w,p) \triangleq \begin{cases} \bot & \text{if } g(s,r) = \bot,\\ (s',r',w \times w', p \times p') & \text{if } g(s,r) = (s',r',w',p'). \end{cases}$$

This lifting lets us compose two functions in $\mathcal{D}$.

Using these preliminary definitions, we define a density semantics in Figure 6, where a command C means a function $[\![C]\!]_d \in \mathcal{D}$. The notation $r \setminus v$ in the figure means the removal of the entry v from the finite map r if $v \in \mathrm{dom}(r)$; otherwise, $r \setminus v$ is just r. The set membership $[\![C]\!]_d \in \mathcal{D}$ says that $[\![C]\!]_d$ is a local measurable function from $\mathit{Store} \times \mathit{RDB}$ to $\{\bot\} \cup \mathit{Store} \times \mathit{RDB} \times [0,\infty) \times [0,\infty)$. Thus, the function $[\![C]\!]_d$ takes a store s and a random database r as inputs, where the former fixes the values of variables at the start of C and the latter specifies random seeds some of which C may consume to sample random variables. Given such inputs, the function outputs an updated store s', the part r' of r not consumed by C, the total score w' expressing how well the execution of C matches observations, and the probability density p' of C at the consumed part of r. If r does not contain enough random seeds, or the execution of C encounters some runtime error, or it falls into an infinite loop, then the function returns $\bot$.

Lemma 4.6. *For every command C, its semantics $[\![C]\!]_d$ is well-defined and belongs to $\mathcal{D}$.*

The density semantics $[\![C]\!]_d$ is closely related to the measure semantics $[\![C]\!]$ defined in §4.2. Both interpretations of C describe the computation of C but from slightly different perspectives. To state this relationship formally, we need a few notations. For $g \in \mathcal{D}$ and $s \in \mathit{Store}$, define

$$\mathit{dens}(g,s) : \mathit{RDB} \to [0,\infty), \qquad \mathit{dens}(g,s)(r) \triangleq \begin{cases} w' \times p' & \text{if } \exists s', w', p'.\ (g(s,r) = (s', [], w', p')), \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathit{get}(g,s) : \mathit{RDB} \to \mathit{Store} \cup \{\bot\}, \qquad \mathit{get}(g,s)(r) \triangleq \begin{cases} s' & \text{if } \exists s', w', p'.\ (g(s,r) = (s', [], w', p')), \\ \bot & \text{otherwise.} \end{cases}$$

Both $\mathit{dens}(g,s)$ and $\mathit{get}(g,s)$ are concerned with random databases r that precisely describe the randomness needed by the execution of g from s. This is formalised by the use of [] in the definitions. The function $\mathit{dens}(g,s)$ assigns a score to such an r, and in so doing, it defines a probability density on RDB with respect to the reference measure ρ. The function $\mathit{get}(g,s)$ computes a store that the computation of g would result in when started with such an r. We often write $\mathit{dens}(C,s)$ and $\mathit{get}(C,s)$ to mean $\mathit{dens}([\![C]\!]_d, s)$ and $\mathit{get}([\![C]\!]_d, s)$, respectively.

Lemma 4.7. *For all $g \in \mathcal{D}$, the following functions from $\mathit{Store} \times \mathit{RDB}$ to $\mathbb{R}$ and $\{\bot\} \cup \mathit{Store}$ are measurable: $(s,r) \longmapsto \mathit{dens}(g,s)(r)$ and $(s,r) \longmapsto \mathit{get}(g,s)(r)$.*

The next lemma is the main reason that we considered the locality property. It plays a crucial role in proving Theorem 4.9, the key result of this subsection.

Lemma 4.8. *For all non-negative bounded measurable functions $h : (\{\bot\} \cup \mathit{Store}) \times \mathit{RDB} \to \mathbb{R}$, stores s, and functions $g_1, g_2 \in \mathcal{D}$, we have that*

$$\begin{aligned} &\int \rho(\mathrm{d}r) \left(\mathit{dens}(g_2^{\ddagger} \circ g_1, s)(r) \cdot h\Big(\mathit{get}(g_2^{\ddagger} \circ g_1, s)(r), r\Big)\right) \\ &= \int \rho(\mathrm{d}r_1) \left(\mathit{dens}(g_1,s)(r_1) \cdot \mathbb{1}_{[\mathit{get}(g_1,s)(r_1) \neq \bot]} \right. \\ &\qquad \left. \cdot \int \rho(\mathrm{d}r_2) \left(\mathit{dens}(g_2, \mathit{get}(g_1,s)(r_1))(r_2) \cdot \mathbb{1}_{[r_1 \# r_2]} \cdot h\Big(\mathit{get}(g_2, \mathit{get}(g_1,s)(r_1))(r_2), r_1 \uplus r_2\Big)\right)\right). \end{aligned}$$

Assume that $(\{\bot\} \cup \mathit{Store}) \times \mathit{RDB}$ is ordered as follows: for all $(a,r), (a',r') \in (\{\bot\} \cup \mathit{Store}) \times \mathit{RDB}$,

$$(a,r) \sqsubseteq (a',r') \iff (a = \bot \vee a = a') \wedge r = r'.$$

Theorem 4.9. *For all non-negative bounded measurable monotone functions $h : (\{\bot\} \cup \mathit{Store}) \times \mathit{RDB} \to \mathbb{R}$ and states σ,*

$$\int [\![C]\!](\sigma)(\mathrm{d}(\sigma', w'))\, (w' \cdot h(\sigma'_s, \sigma'_r)) = \int \rho(\mathrm{d}r') \left(\mathit{dens}(C, \sigma_s)(r') \cdot \mathbb{1}_{[r' \# \sigma_r]} \cdot h(\mathit{get}(C, \sigma_s)(r'), r' \uplus \sigma_r)\right).$$

Corollary 4.10. $\mathrm{Mea}(C, A) = \int \rho(\mathrm{d}r)\, (\mathbb{1}_{[r \in A]} \cdot \mathit{dens}(C, s_I)(r))$ *for all C and all measurable A.*

This corollary says that $\mathit{dens}(C, s_I)$ is the density of the measure $\mathrm{Mea}(C, \cdot)$ with respect to ρ, and supports our informal claim that $[\![C]\!]_d$ computes the density of the measure $[\![C]\!]$.

5 STOCHASTIC VARIATIONAL INFERENCE

In this section, we explain stochastic variational inference (SVI) algorithms using the semantics that we have developed so far. In particular, we describe the requirements made implicitly by one fundamental SVI algorithm, which is regarded most permissive by the ML researchers because the algorithm does not require the differentiability of the density of a given probabilistic model.

We call a command C *model* if it has a finite nonzero normalising constant:

$$Z_C = \mathrm{Mea}(C, \mathit{RDB}) = \left(\int [\![C]\!](\sigma_I)(\mathrm{d}(\sigma, w))\, w \right) \in (0, \infty),$$

$$\begin{aligned}\text{model } C \;\equiv\; & \big(s := \mathsf{sample}_{\mathsf{norm}}(\text{“slope”}, 0.0, 5.0);\; i := \mathsf{sample}_{\mathsf{norm}}(\text{“intercept”}, 0.0, 5.0);\\ & x_1 := 1.0;\; y_1 := 2.3;\; x_2 := 2.0;\; y_2 := 4.2;\; x_3 := 3.0;\; y_3 := 6.9;\\ & \mathsf{score}_{\mathsf{norm}}(y_1, s \cdot x_1 + i, 1.0);\; \mathsf{score}_{\mathsf{norm}}(y_2, s \cdot x_2 + i, 1.0);\; \mathsf{score}_{\mathsf{norm}}(y_3, s \cdot x_3 + i, 1.0)\big)\\ \text{guide } D_\theta \;\equiv\; & \big(s := \mathsf{sample}_{\mathsf{norm}}(\text{“slope”}, \theta_1, \exp(\theta_2));\; i := \mathsf{sample}_{\mathsf{norm}}(\text{“intercept”}, \theta_3, \exp(\theta_4))\big)\end{aligned}$$

Fig. 7. Example model-guide pair for simple Bayesian linear regression.

where σ_I is the initial state. Given a model C, the SVI algorithms attempt to infer a good approximation of C's posterior distribution $\Pr(C, \cdot)$ defined in (2). They tackle this posterior-inference problem in two steps.

First, the SVI algorithms fix a collection of approximate distributions. They usually do so by asking the developer of C to provide a command D_θ parameterised by $\theta \in \mathbb{R}^p$, which can serve as a template for approximation distributions. The command D_θ typically has a control-flow structure similar to that of C, but it is simpler than C: it does not use any score statements, and may replace complex computation steps of C by simpler ones. In fact, D_θ should satisfy two formal requirements, which enforce this simplicity. The first is

$$\mathrm{Mea}(D_\theta, \mathit{RDB}) = 1 \quad \text{for all } \theta \in \mathbb{R}^p,$$

which means that the normalising constant of D_θ is 1. The second is that D_θ should keep the score (i.e., the w component) to be 1, i.e.,

$$[\![D_\theta]\!](\sigma_I)(\mathit{State} \times ([0, \infty) \setminus \{1\})) = 0.$$

Meeting these requirements is often not too difficult. A common technique is to ensure that D_θ does not use the score statement and always terminates. Figure 7 gives an example of (C, D_θ) for simple Bayesian linear regression with three data points. Note that in this case, D_θ is obtained from C by deleting the score statements and replacing the arguments 0.0 and 5.0 of normal distributions by parameter $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$. Following the terminology of Pyro, we call a parameterised command D_θ *guide* if it satisfies the two requirements just mentioned.

Second, the SVI algorithms search for a good parameter θ that makes the distribution described by D_θ close to the posterior of C. Concretely, they formulate an optimisation problem where the optimisation objective expresses that some form of distance from D_θ's distribution to C's posterior should be minimised. Then, they solve the problem by a version of gradient descent.

The KL divergence is a standard choice for distance. Let μ, μ' be measures on RDB that have densities g and g' with respect to the measure ρ. The KL divergence from g to g' is defined by

$$\mathrm{KL}(g||g') \triangleq \int \rho(\mathrm{d}r) \left(g(r) \cdot \log \frac{g(r)}{g'(r)} \right). \tag{4}$$

In words, it is the ratio of densities g and g' averaged according to g. If $g = g'$, the ratio is always 1, so that the KL becomes 0. The KL divergence is defined only if the following conditions are met:

- *Absolute continuity*: $g'(r) = 0 \implies g(r) = 0$ for all $r \in \mathit{RDB}$,[9] which ensures that the integrand in (4) is well-defined even when the denominator $g'(r)$ in (4) takes the value 0;
- *Integrability*: the integral in (4) has a finite value.

Using our semantics, we can express the KL objective as follows:

$$\mathrm{argmin}_{\theta \in \mathbb{R}^p} \mathrm{KL}\left(\mathit{dens}(D_\theta, s_I) \middle\| \frac{\mathit{dens}(C, s_I)}{Z_C} \right). \tag{5}$$

[9] This condition can be relaxed in a more general formulation of the KL divergence stated in terms of the so called Radon-Nikodym derivative. We do not use the relaxed condition to reduce the amount of materials on measure theory in the paper.

Recall that $dens(D_\theta, s_I)$ and $dens(C, s_I)/Z_C$ are densities of the probability measures of the command D_θ and the posterior of C (Corollary 4.10), and they are defined by means of our density semantics in §4.3. Most SVI engines solve this optimisation problem by a version of gradient descent.

In the paper, we consider one of the most fundamental and versatile SVI algorithms. The algorithm is called score estimator or REINFORCE, and it works by estimating the gradient of the objective in (5) using samples and performing the gradient descent with this estimated gradient. More concretely, the algorithm starts by initialising θ with some value (usually chosen randomly) and updating it repeatedly by the following procedure:

(i) Sample $r_1, \ldots, r_N$ independently from $dens(D_\theta, s_I)$

(ii) $\theta \leftarrow \theta - \eta \times \left(\frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log dens(D_\theta, s_I)(r_i) \right) \cdot \log \frac{dens(D_\theta, s_I)(r_i)}{dens(C, s_I)(r_i)} \right)$

Here N and η are hyperparameters to this algorithm, the former determining the number of samples used to estimate the gradient and the latter, called learning rate, deciding how much the algorithm should follow the direction of the estimated gradient. Although we do not explain here, sampling $r_1, \ldots, r_N$ and computing all of $dens(D_\theta, s_I)(r_i)$, $dens(C, s_I)(r_i)$ and $\nabla_\theta(\log dens(D_\theta, s_I)(r_i))$ can be done by executing D_θ and C multiple times under slightly unusual operational semantics [Yang 2019]. The SVI engines of Pyro and Anglican implement such operational semantics.

The average over the N terms in the θ-update step is the core of the algorithm. It approximates the gradient of the optimisation objective in (5):

$$\nabla_\theta \mathrm{KL}\left(dens(D_\theta, s_I) \middle\| \frac{dens(C, s_I)}{Z_C} \right) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log dens(D_\theta, s_I)(r_i) \right) \cdot \log \frac{dens(D_\theta, s_I)(r_i)}{dens(C, s_I)(r_i)}.$$

The average satisfies an important property called *unbiasedness*, summarised by Theorem 5.1.

Theorem 5.1. *Let C be a model, D_θ be a guide, and $N \neq 0 \in \mathbb{N}$. Define $\mathrm{KL}_{(-)} : \mathbb{R}^p \to \mathbb{R}_{\geq 0}$ as $\mathrm{KL}_\theta \triangleq \mathrm{KL}(dens(D_\theta, s_I) || dens(C, s_I)/Z_C)$. Then, $\mathrm{KL}_{(-)}$ is well-defined and continuously differentiable with*

$$\nabla_\theta \mathrm{KL}_\theta = \mathbb{E}_{\prod_i dens(D_\theta, s_I)(r_i)} \left[\frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log dens(D_\theta, s_I)(r_i) \right) \log \frac{dens(D_\theta, s_I)(r_i)}{dens(C, s_I)(r_i)} \right] \tag{6}$$

if

(R1) $dens(C, s_I)(r) = 0 \implies dens(D_\theta, s_I)(r) = 0$, for all $r \in RDB$ and $\theta \in \mathbb{R}^p$;

(R2) for all $(r, \theta, j) \in RDB \times \mathbb{R}^p \times [p]$, the function $v \longmapsto dens(D_{\theta[j:v]}, s_I)(r)$ on $\mathbb{R}$ is differentiable;

(R3) for all $\theta \in \mathbb{R}^p$,

$$\int \rho(\mathrm{d}r) \left(dens(D_\theta, s_I)(r) \cdot \log \frac{dens(D_\theta, s_I)(r)}{dens(C, s_I)(r)} \right) < \infty;$$

(R4) for all $(\theta, j) \in \mathbb{R}^p \times [p]$, the function

$$v \longmapsto \int \rho(\mathrm{d}r) \left(dens(D_{\theta[j:v]}, s_I)(r) \cdot \log \frac{dens(D_{\theta[j:v]}, s_I)(r)}{dens(C, s_I)(r)} \right)$$

on $\mathbb{R}$ is continuously differentiable;

(R5) for all $\theta \in \mathbb{R}^p$,

$$\nabla_\theta \int \rho(\mathrm{d}r) \left(dens(D_\theta, s_I)(r) \cdot \log \frac{dens(D_\theta, s_I)(r)}{dens(C, s_I)(r)} \right) = \int \rho(\mathrm{d}r)\, \nabla_\theta \left(dens(D_\theta, s_I)(r) \cdot \log \frac{dens(D_\theta, s_I)(r)}{dens(C, s_I)(r)} \right);$$

(R6) for all $\theta \in \mathbb{R}^p$,

$$\int \rho(\mathrm{d}r)\, \nabla_\theta\, dens(D_\theta, s_I)(r) = \nabla_\theta \int \rho(\mathrm{d}r)\, dens(D_\theta, s_I)(r).$$

Here $\theta[j : v]$ denotes a vector in $\mathbb{R}^p$ that is the same as θ except that its j-th component is v.

The conclusion of this theorem (Equation (6)) and its proof are well-known [Ranganath et al. 2014], but the requirements in the theorem (and the continuous differentiability of KL_θ in the conclusion) are rarely stated explicitly in the literature.

The correctness of the algorithm crucially relies on the unbiasedness property in Theorem 5.1. The property ensures that the algorithm converges to a local minimum with probability 1. Thus, it is important that the requirements in the theorem are met. In fact, some of the requirements there are needed even to state the optimisation objective in (5), because without them, the objective does not exist. In the next section, we describe conditions that imply those requirements and can serve as target properties of program analysis for probabilistic programs. The latter point is worked out in detail in §7 and §8 where we discuss program analysis for probabilistic programs and SVI.

6 CONDITIONS FOR STOCHASTIC VARIATIONAL INFERENCE

Ideally we want to have program analysers that discharge the six requirements R1-R6 in Theorem 5.1. However, except R1 and R2, the requirements are not ready for serving as the targets of static analysis algorithms. Automatically discharging them based on the first principles (such as the definition of integrability with respect to a measure) may be possible, but seems less immediate than doing so using powerful theorems from continuous mathematics.

In this section, we explain conditions that imply the requirements R3-R6 and are more friendly to program analysis than the requirements themselves. The conditions are given in two boxes (9) and (10). Throughout the section, we fix a model C and a guide D_θ.

6.1 Assumption

Throughout the section, we assume that the densities of D_θ and C have the following form:

$$dens(D_\theta, s_I)(r) = \sum_{i=1}^{M} \mathbb{1}_{[r\in A_i]} \prod_{\alpha\in K_i} \mathcal{N}\big(r(\alpha); \mu_{(i,\alpha)}(\theta), \sigma_{(i,\alpha)}(\theta)\big), \tag{7}$$

$$dens(C, s_I)(r) = \sum_{i=1}^{M} \mathbb{1}_{[r\in A_i]} \left(\prod_{\alpha\in K_i} \mathcal{N}\big(r(\alpha); \mu'_{(i,\alpha)}(r), \sigma'_{(i,\alpha)}(r)\big)\right)\left(\prod_{j\in[N_i]} \mathcal{N}\big(c_{(i,j)}; \mu''_{(i,j)}(r), \sigma''_{(i,j)}(r)\big)\right),$$

where

- $M, N_i \in \mathbb{N} \setminus \{0\}$;
- $A_1, \ldots, A_M$ are disjoint measurable subsets of RDB;
- K_i's are finite sets of strings such that $\mathrm{dom}(r) = K_i$ for all $r \in A_i$;
- $\mu_{(i,\alpha)}$ and $\sigma_{(i,\alpha)}$ are measurable functions from $\mathbb{R}^p$ to $\mathbb{R}$ and $(0,\infty)$, respectively;
- $\mu'_{(i,\alpha)}$ and $\mu''_{(i,j)}$ are measurable functions from $[K_i \to \mathbb{R}]$ to $\mathbb{R}$;
- $\sigma'_{(i,\alpha)}$ and $\sigma''_{(i,j)}$ are measurable functions from $[K_i \to \mathbb{R}]$ to $(0,\infty)$;
- $c_{(i,j)}$ is a real number.

In programming terms, our assumption first implies that both C and D_θ use at most a fixed number of random variables. That is, the number of random variables they generate must be finite not only within a single execution but also across all possible executions since the names of all random variables are found in a finite set $\bigcup_{i=1}^{M} K_i$. This property is met if the number of steps in every execution of C and D_θ from σ_I is bounded by some T and the executions over the same program path use the same set of random variables. Note that the bound may depend on s_I. Such a bound exists for most probabilistic programs.[10] Also, the assumption says that the parameters of normal distributions in sample statements in D_θ may depend only on θ, but not on other sampled random

[10]Notable exceptions are models using probabilistic grammars or those from Bayesian nonparametrics.

variables. This is closely related to a common approach for designing approximate distributions in variational inference, called mean-field approximation, where the approximate distribution consists of independent normal random variables.

We use the term "assumption" here, instead of "condition" in the following subsections because the assumed properties are rather conventional and they are not directly related to the requirements in Theorem 5.1, at least not as much as the conditions that we will describe next.

6.2 Condition for Requirement R3

Note that the integral in the requirement R3 can be written as the sum of two expectations:

$$\begin{aligned}&\int \rho(\mathrm{d}r)\left(dens(D_\theta, s_I)(r)\cdot\log\frac{dens(D_\theta, s_I)(r)}{dens(C, s_I)(r)}\right)\\&= \mathbb{E}_{dens(D_\theta,s_I)(r)}\left[\log dens(D_\theta, s_I)(r)\right] - \mathbb{E}_{dens(D_\theta,s_I)(r)}\left[\log dens(C, s_I)(r)\right].\end{aligned}\tag{8}$$

The minus of the first term (i.e., $-\mathbb{E}_{dens(D_\theta,s_I)(r)}\left[\log dens(D_\theta, s_I)(r)\right]$) is called the *differential entropy* of the density $dens(D_\theta, s_I)$. Intuitively, it is large when the density on $\mathbb{R}^n$ is close to the Lebesgue measure, which is regarded to represent the absence of information. The differential entropy is sometimes undefined [Ghourchian et al. 2017]. Fortunately, a large class of probability densities (containing many commonly used probability distributions) have well-defined entropies [Ghourchian et al. 2017; Nair et al. 2006]. Our $dens(D_\theta, s_I)$ is one of such fortunate cases.

Theorem 6.1. $\mathbb{E}_{dens(D_\theta,s_I)(r)}\left[|\log dens(D_\theta, s_I)(r)|\right] < \infty$ *under our assumption in §6.1.*

We remark that a violation of the assumption (7) for D_θ can result in an undefined entropy, as illustrated by the following examples.

Example 6.2. Consider guides $D_{(i,\theta)}$ defined as follows ($i = 1, 2$):

$$D_{(i,\theta)} \equiv (x_1 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_1\text{”}, \theta_1, 1);\ x_2 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_2\text{”}, \theta_2, E_i[x_1]))$$

where for some $n \geq 1$ and $c \neq 0 \in \mathbb{R}$,[11]

$$E_1[x_1] \equiv \mathsf{if}\ (x_1{=}0)\ \mathsf{then}\ 1\ \mathsf{else}\ \exp(-1/|x_1|^n), \qquad E_2[x_1] \equiv \exp(\exp(c\cdot x_1^3)).$$

None of $dens(D_{(i,\theta)}, s_I)$'s satisfies the assumption (7) because the standard deviation of the normal distribution for x_2 depends on the value of x_1. The entropies of $dens(D_{(i,\theta)}, s_I)$'s are all undefined: $\mathbb{E}_{dens(D_{(i,\theta)},s_I)(r)}[|\log dens(D_{(i,\theta)}, s_I)(r)|] = \infty$ for all $i = 1, 2$. □

Since the first term of (8) is always finite by Theorem 6.1, it is enough to ensure the finiteness of the second term of (8). For $i \in [M]$, define the set of (absolute) affine functions on $[K_i \to \mathbb{R}]$ as:

$$\mathcal{A}_i \triangleq \left\{ f \in [[K_i \to \mathbb{R}] \to \mathbb{R}] \;\middle|\; f(r) = c + \sum_{\alpha\in K_i} (c_\alpha \cdot |r(\alpha)|) \ \text{ for some } c, c_\alpha \in \mathbb{R} \right\}.$$

Our condition for ensuring the finiteness of the second term is as follows:

$$\boxed{\begin{array}{l}\text{For all } i \in [M], \text{ there are } f', f'', l', u', l'', u'' \in \mathcal{A}_i \text{ such that}\\ \quad |\mu'_{(i,\alpha)}(r)| \leq \exp(f'(r)), \qquad \exp(l'(r)) \leq \sigma'_{(i,\alpha)}(r) \leq \exp(u'(r)),\\ \quad |\mu''_{(i,j)}(r)| \leq \exp(f''(r)), \qquad \exp(l''(r)) \leq \sigma''_{(i,j)}(r) \leq \exp(u''(r)),\\ \text{all four hold for every } (\alpha, j, r) \in K_i \times [N_i] \times A_i.\end{array}}\tag{9}$$

Theorem 6.3. *The condition (9) implies* $\mathbb{E}_{dens(D_\theta,s_I)(r)}\left[|\log dens(C, s_I)(r)|\right] < \infty$ *under our assumption in §6.1. Thus, in that case, it entails the requirement R3 (i.e., the objective in (5) is well-defined).*

[11] Formally, we should implement $E_1[x_1]$ as an application of a primitive function f_1 to x_1 that has the semantics described by the if-then-else statement here.

Our condition in (9) is sufficient but not necessary for the objective in (5) to be defined. However, its violation is a good warning, as illustrated by our next examples.

Example 6.4. Consider models $C_1, \ldots, C_4$ and a guide D_θ defined as follows:

$$\begin{aligned}
C_i &\equiv (x_1 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_1\text{”}, 0, 1);\ x_2 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_2\text{”}, E_i[x_1], 1)) \quad \text{for } i = 1, 2\\
C_i &\equiv (x_1 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_1\text{”}, 0, 1);\ x_2 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_2\text{”}, 0, E_i[x_1])) \quad \text{for } i = 3, 4\\
D_\theta &\equiv (x_1 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_1\text{”}, \theta_1, 1);\ x_2 := \mathsf{sample}_{\mathsf{norm}}(\text{“}a_2\text{”}, \theta_2, 1))
\end{aligned}$$

where for some $n \geq 1$ and $c \neq 0 \in \mathbb{R}$,

$$E_1[x_1] \equiv \mathsf{if}\ (x_1{=}0)\ \mathsf{then}\ 0\ \mathsf{else}\ \frac{1}{x_1^n}, \quad E_2[x_1] \equiv E_4[x_1] \equiv \exp(c{\cdot}x_1^3), \quad E_3[x_1] \equiv \mathsf{if}\ (x_1{=}0)\ \mathsf{then}\ 1\ \mathsf{else}\ |x_1|^n.$$

Let $A = [\{\text{“}a_1\text{”}, \text{“}a_2\text{”}\} \to \mathbb{R}]$. For $r \in A$, define

$$\begin{aligned}
\mu_1'(r) &\triangleq \mathsf{if}\ r(\text{“}a_1\text{”}) = 0\ \mathsf{then}\ 0\ \mathsf{else}\ 1/r(\text{“}a_1\text{”})^n, & \mu_2'(r) &\triangleq \exp(c \cdot r(\text{“}a_1\text{”})^3),\\
\sigma_3'(r) &\triangleq \mathsf{if}\ r(\text{“}a_1\text{”}) = 0\ \mathsf{then}\ 1\ \mathsf{else}\ |r(\text{“}a_1\text{”})|^n, & \sigma_4'(r) &\triangleq \exp(c \cdot r(\text{“}a_1\text{”})^3).
\end{aligned}$$

Then, we have that

$$\begin{aligned}
\mathit{dens}(C_i, s_I)(r) &= \mathbb{1}_{[r \in A]} \cdot \mathcal{N}(r(\text{“}a_1\text{”}); 0, 1) \cdot \mathcal{N}(r(\text{“}a_2\text{”}); \mu_i'(r), 1) \quad \text{for } i = 1, 2\\
\mathit{dens}(C_i, s_I)(r) &= \mathbb{1}_{[r \in A]} \cdot \mathcal{N}(r(\text{“}a_1\text{”}); 0, 1) \cdot \mathcal{N}(r(\text{“}a_2\text{”}); 0, \sigma_i'(r)) \quad \text{for } i = 3, 4\\
\mathit{dens}(D_\theta, s_I)(r) &= \mathbb{1}_{[r \in A]} \cdot \mathcal{N}(r(\text{“}a_1\text{”}); \theta_1, 1) \cdot \mathcal{N}(r(\text{“}a_2\text{”}); \theta_2, 1).
\end{aligned}$$

None of μ_1', μ_2', σ_3', and σ_4' satisfies the condition in (9). The function μ_1' is not bounded in $\{r \in A \mid -1 \leq r(\text{“}a_1\text{”}) \leq 1 \wedge -1 \leq r(\text{“}a_2\text{”}) \leq 1\}$, but every μ' satisfying the condition in (9) should be bounded. Also, the cubic exponential growth of μ_2' cannot be bounded by any linear exponential function on r. The violation of the condition by σ_3' and σ_4' can be shown similarly.

In fact, the objective in (5) is not defined for all of the four cases. This is because for all $i = 1, \ldots, 4$, $\mathbb{E}_{\mathit{dens}(D_\theta, s_I)(r)}[|\log \mathit{dens}(C_i, s_I)(r)|] = \infty$. □

We now show that the condition in (9) is satisfied by a large class of functions in machine learning applications, including functions using neural networks. We call a function $nn : \mathbb{R}^n \to \mathbb{R}$ an *affine-bounded neural network* if there exist functions $f_j : \mathbb{R}^{n_j} \to \mathbb{R}^{n_{j+1}}$ and affine functions $l_j : \mathbb{R}^{n_j} \to \mathbb{R}$ for all $1 \leq j \leq d$ such that (i) $n_1 = n$ and $n_{d+1} = 1$; (ii) $nn = f_d \circ \cdots \circ f_1$; and (iii) $\|f_j(v)\|_1 \leq l_j(|v_1|, \ldots, |v_{n_j}|)$ for all $1 \leq j \leq d$ and $v \in \mathbb{R}^{n_j}$, where $\|\cdot\|_1$ denotes the ℓ_1-norm. Note that each component of f_j can be, for instance, an affine function, one of commonly used activation functions (e.g., relu, tanh, sigmoid, softplus), or one of min/max functions (min and max). Therefore, most of neural networks used in machine learning applications are indeed affine-bounded. Lemma 6.5 indicates that a wide range of functions satisfy the condition (9).

Lemma 6.5. *Pick $i \in [M]$ and $\alpha \in K_i$. Let $(\alpha_1, \ldots, \alpha_J)$ be an enumeration of the elements in K_i, and $\overline{r} \triangleq (r(\alpha_1), \ldots, r(\alpha_J)) \in \mathbb{R}^J$ be an enumeration of the values of $r \in A_i$. Consider any affine-bounded neural network $nn : \mathbb{R}^J \to \mathbb{R}$, polynomial $poly : \mathbb{R}^J \to \mathbb{R}$, and $c \in (0, \infty)$. Then, the below list of functions $\mu_{(i,\alpha)}'$ and $\sigma_{(i,\alpha)}'$ on A_i satisfy the condition in (9):*

$$\begin{aligned}
\mu_{(i,\alpha)}'(r) &= nn(\overline{r}), & \mu_{(i,\alpha)}'(r) &= poly(\overline{r}), & \mu_{(i,\alpha)}'(r) &= \exp(nn(\overline{r})),\\
\sigma_{(i,\alpha)}'(r) &= |nn(\overline{r})| + c, & \sigma_{(i,\alpha)}'(r) &= |poly(\overline{r})| + c, & \sigma_{(i,\alpha)}'(r) &= \exp(nn(\overline{r})),\\
\sigma_{(i,\alpha)}'(r) &= (|nn(\overline{r})| + c)^{-1}, & \sigma_{(i,\alpha)}'(r) &= (|poly(\overline{r})| + c)^{-1}, & \sigma_{(i,\alpha)}'(r) &= \mathrm{softplus}(nn(\overline{r})),
\end{aligned}$$

where $\mathrm{softplus}(v) \triangleq \log(1 + \exp(v))$. *Moreover, the same holds for $\mu_{(i,j)}''$ and $\sigma_{(i,j)}''$ as well.*

6.3 Condition for Requirements R4-R6

Assume that the model C and the guide D_θ satisfy our assumption and condition in the previous two subsections. Our condition for the requirements R4-R6 is given below:

$$\boxed{\begin{array}{l}\text{For all } i \in [M],\ \alpha \in K_i, \text{ and } (\theta, j) \in \mathbb{R}^p \times [p],\\ \quad \text{the function } v \in \mathbb{R} \longmapsto \mu_{(i,\alpha)}(\theta[j:v]) \text{ is continuously differentiable;}\\ \quad \text{the function } v \in \mathbb{R} \longmapsto \sigma_{(i,\alpha)}(\theta[j:v]) \text{ is continuously differentiable.}\end{array}} \tag{10}$$

Theorem 6.6. *If both our assumption in §6.1 and the condition (9) hold, then the condition (10) implies the requirements R4-R6.*

The proof of the theorem uses the following nontrivial result [Klenke 2014, Theorem 6.28] about exchanging differentiation and integration, a consequence of the dominated convergence theorem.

Theorem 6.7. *Let $V \subset \mathbb{R}$ be an open interval, and (X, Σ, μ) be a measure space. Suppose that a measurable function $f : V \times X \to \mathbb{R}$ satisfies the following conditions: (a) for all $v \in V$, the integral $\int \mu(\mathrm{d}x)\, f_v(x)$ is well-defined; (b) for almost all $x \in X$ (w.r.t. μ) and all $v \in V$, the partial derivative $\nabla_v f_v(x)$ with respect to v is well-defined;[12] (c) there is a measurable function $h : X \to \mathbb{R}$ such that $\int \mu(\mathrm{d}x)\, h(x)$ is well-defined and $|\nabla_v f_v(x)| \le h(x)$ for all $v \in V$ and almost all $x \in X$ (w.r.t. μ). Then, for all $v \in V$, both sides of the below equation are well-defined, and the equality holds:*

$$\nabla_v \int \mu(\mathrm{d}x)\, f_v(x) = \int \mu(\mathrm{d}x)\, \nabla_v f_v(x).$$

Note that the theorem ensures not only the validity of interchanging differentiation and integration, but also the differentiability of $\int \mu(\mathrm{d}x)\, f_v(x)$ (w.r.t. v) and the integrability of $\nabla_v f_v(x)$ over $x \in X$.

Our condition in (10) is sufficient but not necessary for the requirements R4-R6 to hold, in particular for the objective in (5) to have well-defined partial derivatives in θ. However, its violation is a good indication of a potential problem. The following example illustrates this point.

Example 6.8. Consider a model C and a guide D_θ defined as follows:

$$C \equiv x := \mathsf{sample}_{\mathsf{norm}}(\text{“}a\text{”}, 0, 1) \qquad D_\theta \equiv x := \mathsf{sample}_{\mathsf{norm}}(\text{“}a\text{”}, 0, E[\theta])$$

where $E[\theta] \equiv \mathrm{relu}(\theta)+2$ and $\mathrm{relu}(v) \triangleq \mathbb{1}_{[v \ge 0]} \cdot v$. Such $E[\theta]$ can definitely appear in machine learning applications, once a guide starts to use neural networks with parameters θ. Let $A = [\{\text{“}a\text{”}\} \to \mathbb{R}]$ and $\sigma(\theta) \triangleq \mathrm{relu}(\theta) + 2$. Then, $dens(C, s_I)(r) = \mathbb{1}_{[r \in A]} \cdot \mathcal{N}(r(\text{“}a\text{”}); 0, 1)$ and $dens(D_\theta, s_I)(r) = \mathbb{1}_{[r \in A]} \cdot \mathcal{N}(r(\text{“}a\text{”}); 0, \sigma(\theta))$. Note (10) is violated: σ is non-differentiable at $\theta = 0$. A simple calculation shows:

$$\nabla_\theta \int \rho(\mathrm{d}r) \left(dens(D_\theta, s_I)(r) \cdot \log \frac{dens(D_\theta, s_I)(r)}{dens(C, s_I)(r)} \right) = \begin{cases} 0 & \text{if } \theta \in (-\infty, 0) \\ ((2+\theta)^2 - 1)/(2+\theta) & \text{if } \theta \in (0, \infty) \\ \text{undefined} & \text{if } \theta = 0. \end{cases}$$

Hence, the objective in (5) does not have a well-defined partial derivative in θ at $\theta = 0$. □

7 ANALYSIS

In this section, we describe a recipe for building a static analysis that automatically discharges some of the assumptions and conditions given in §6. The recipe ensures that the constructed static analyses are sound with respect to the density semantics in §4.3. We illustrate it by describing four static analyses for verifying the model-guide support match (the requirement R1), the guide-parameter differentiability (the requirement R2), the condition (9), and the condition (10). The analysis for the model-guide support match has been developed significantly more for the Pyro

[12] A more popular notation is $(\partial f_v(x))/(\partial v)$, but we opt for $\nabla_v f_v(x)$ to avoid clutter.

$$\begin{aligned}
[\![\mathsf{skip}]\!]^\sharp &\triangleq \mathit{skip}^\sharp & [\![\mathsf{if}\ E\ \{C_0\}\ \mathsf{else}\ \{C_1\}]\!]^\sharp &\triangleq \mathit{cond}(E)^\sharp([\![C_0]\!]^\sharp, [\![C_1]\!]^\sharp)\\
[\![x := E]\!]^\sharp &\triangleq \mathit{update}(x, E)^\sharp & [\![x := \mathsf{sample}_{\mathsf{norm}}(S, E_1, E_2)]\!]^\sharp &\triangleq \mathit{sample}(x, S, E_1, E_2)^\sharp\\
[\![C_0; C_1]\!]^\sharp &\triangleq [\![C_1]\!]^\sharp \circ^\sharp [\![C_0]\!]^\sharp & [\![\mathsf{score}_{\mathsf{norm}}(E_0, E_1, E_2)]\!]^\sharp &\triangleq \mathit{score}(E_0, E_1, E_2)^\sharp\\
[\![\mathsf{while}\ E\ \{C\}]\!]^\sharp &\triangleq (\mathit{wfix}\, T)\ (\text{where } T(t') \triangleq \mathit{cond}(E)^\sharp(t' \circ^\sharp [\![C]\!]^\sharp,\ \mathit{skip}^\sharp))
\end{aligned}$$

Fig. 8. Abstract semantics $[\![C]\!]^\sharp \in \mathcal{T}^\sharp$ of commands C

programming language, and applied to analyse realistic examples of the language. This fully-blown analysis and our experiments will be described in §8.

Throughout this section, we assume that the parameters θ of a guide D_θ are included in Var, and are only read by D_θ and not accessed by a model C. What we used to call s_I will be the part of the store for variables in $\mathit{Var} \setminus \theta$, and what we used to write θ will correspond to the other part for θ.

7.1 Generic Program Analysis Framework

Our recipe is for building a static analysis that infers information about the state transformation of a given command. It is similar to the conventional methodology for building a so called *relational* static analysis, which also attempts to find information about the relationship between input and output states of a given command. However, our recipe diverges from the convention in one important point: while the abstract states of conventional relational analyses represent relations on states, we let abstract states directly express sets of concrete state transformers. This departure from the convention is due to the difficulty of using relations for expressing properties of state transformers that we desire. For instance, we could not express a set of functions with a certain type of differentiability using relations.

Recall the domain $\mathcal{D}$ in (3), and the notion of *admissible* subset from domain theory: $D_0 \subseteq \mathcal{D}$ is *admissible* if it contains $\bot$ and is closed under taking the limits of ω-chains in D_0.

Our recipe assumes an abstraction instance defined by the following items:

- An *abstract domain*, i.e., a set $\mathcal{T}^\sharp$ with a designated element $\bot^\sharp$.
- A *concretisation function*, i.e., a function $\gamma : \mathcal{T}^\sharp \to \mathcal{P}(\mathcal{D})$ such that for every $t \in \mathcal{T}^\sharp$, $\gamma(t)$ is an admissible subset of $\mathcal{D}$. Note that the concretisation interprets each abstract element t as a set of concrete transformers in $\mathcal{D}$. The admissibility is imposed to enable the sound analysis of loops.
- A *widening operator* $\mathit{widen} : \mathcal{T}^\sharp \times \mathcal{T}^\sharp \to \mathcal{T}^\sharp$, such that for all $t_1, t_2 \in \mathcal{T}^\sharp$ and $i \in [2]$, $\gamma(t_i) \subseteq \gamma(\mathit{widen}(t_1, t_2))$ and for every sequence $\{t_n\}_{n\geq 1}$ in $\mathcal{T}^\sharp$, its widened sequence $\{t'_n\}_{n\geq 1}$, defined by $t'_1 \triangleq t_1$ and $t'_{n+1} \triangleq \mathit{widen}(t'_n, t_{n+1})$ for $n \geq 1$, has an index m such that $t'_m = t'_{m+1}$.
- An *abstract conditional operator* for every expression E, that is, a function $\mathit{cond}(E)^\sharp : \mathcal{T}^\sharp \times \mathcal{T}^\sharp \to \mathcal{T}^\sharp$ such that for all $t_1, t_2 \in \mathcal{T}^\sharp$ and $g_1, g_2 \in \mathcal{D}$, if $g_1 \in \gamma(t_1)$ and $g_2 \in \gamma(t_2)$, then $\big(\lambda(s, r).(\text{if } ([\![E]\!]s{=}\mathit{true}) \text{ then } g_1(s, r) \text{ else } g_2(s, r))\big) \in \gamma(\mathit{cond}(E)^\sharp(t_1, t_2))$.
- An *abstract composition operator* $\circ^\sharp : \mathcal{T}^\sharp \times \mathcal{T}^\sharp \to \mathcal{T}^\sharp$ such that for all $t_1, t_2 \in \mathcal{T}^\sharp$ and $g_1, g_2 \in \mathcal{D}$, if $g_1 \in \gamma(t_1)$ and $g_2 \in \gamma(t_2)$, then $g_2^\ddagger \circ g_1 \in \gamma(t_2 \circ^\sharp t_1)$.
- For all expressions E_0, E_1, E_2 and for all variables x, the abstract elements $\mathit{skip}^\sharp$, $\mathit{update}(x, E_0)^\sharp$, $\mathit{sample}(x, S, E_1, E_2)^\sharp$, and $\mathit{score}(E_0, E_1, E_2)^\sharp \in \mathcal{T}^\sharp$ such that

$$\begin{aligned}
[\![\mathsf{skip}]\!]_d &\in \gamma(\mathit{skip}^\sharp), & [\![x := \mathsf{sample}_{\mathsf{norm}}(S, E_1, E_2)]\!]_d &\in \gamma(\mathit{sample}(x, S, E_1, E_2)^\sharp),\\
[\![x := E_0]\!]_d &\in \gamma(\mathit{update}(x, E_0)^\sharp), & [\![\mathsf{score}_{\mathsf{norm}}(E_0, E_1, E_2)]\!]_d &\in \gamma(\mathit{score}(E_0, E_1, E_2)^\sharp).
\end{aligned}$$

Given these data, we define the static analysis $[\![C]\!]^\sharp \in \mathcal{T}^\sharp$ of a command C in Figure 8. Here the $(\mathit{wfix}\ T)$ is the usual widened fixed point of T, which is defined as the first element t_m with $t_m = t_{m+1}$ in the widened sequence $(t_n)_{n\geq 1}$ where $t_1 \triangleq \bot^\sharp$ and $t_{n+1} \triangleq \mathit{widen}(t_n, T(t_n))$.

Theorem 7.1 (Soundness). *For all commands C, we have $[\![C]\!]_d \in \gamma([\![C]\!]^\sharp)$.*

In the rest of this section, we instantiate this framework into four static analysis instances. In each case, we describe the abstract domain, the abstract bottom element, and the concretisation function. Moreover, in the first two cases, we detail the transfer functions. In the following, for a tuple $(s', r', w', p') \in \mathit{Store} \times \mathit{RDB} \times [0,\infty) \times [0,\infty)$, we use the subscripts $-_s$, $-_r$, $-_w$, and $-_p$ to denote its components. For instance, $(s', r', w', p')_s = s'$ and $(s', r', w', p')_r = r'$.

7.2 Analysis for Model-Guide Match

The first instance analysis finds information about the names of sampled random variables. Such information can be used for discharging the requirement R1, the correspondence between the support of a model and that of a guide. The analysis is based on the below abstraction:

$$\mathcal{T}^\sharp \triangleq \{\bot^\sharp, \top^\sharp\} \cup \mathcal{P}(\mathit{Str}), \qquad \bot^\sharp \triangleq \bot^\sharp, \qquad \gamma(\bot^\sharp) \triangleq \{\lambda(s,r).\,\bot\},$$
$$\gamma(\top^\sharp) \triangleq \mathcal{D}, \qquad \gamma(K) \triangleq \left\{ g \in \mathcal{D} \;\middle|\; \forall s, r.\ g(s,r) \neq \bot \wedge (g(s,r))_r = [] \implies \mathrm{dom}(r) = K \right\},$$

where [] denotes the empty random database. A typical abstract element in $\mathcal{T}^\sharp$ is a set of names K, which represents concrete commands sampling random variables in K. The domain $\mathcal{T}^\sharp$ contains $\bot^\sharp$ and $\top^\sharp$ to express two extreme cases, the set containing only one command that always returns $\bot$, and the set of all commands.

Sound abstract operations can be derived from the density semantics and from the abstraction following the abstract interpretation methodology [Cousot and Cousot 1977]:

$$\mathit{widen}(\bot^\sharp, S) = \mathit{widen}(S, \bot^\sharp) = S; \qquad \mathit{cond}(E)^\sharp = \mathit{widen}; \qquad \mathit{skip}^\sharp = \mathit{update}(x, E_0)^\sharp = \emptyset;$$
$$\mathit{widen}(\top^\sharp, S) = \mathit{widen}(S, \top^\sharp) = \top^\sharp; \qquad \circ^\sharp = \mathit{widen}; \qquad \mathit{sample}(x, S, E_1, E_2)^\sharp = \{x\};$$
$$\mathit{widen}(S_0, S_1) = S_0 \text{ if } S_0 = S_1,\ \top^\sharp \text{ otherwise} \quad \text{for } S_0, S_1 \in \mathcal{P}(\mathit{Str}); \qquad \mathit{score}(E_0, E_1, E_2)^\sharp = \emptyset.$$

We can use the resulting analysis to discharge the requirement R1. We just need to run it on both C and D_θ, and check whether $[\![C]\!]^\sharp = [\![D_\theta]\!]^\sharp = K$ for some $K \in \mathcal{P}(\mathit{Str})$. The positive answer implies the requirement R1, because all the random variables are drawn from the normal distribution. Our extension of this analysis for Pyro (§8) does not rely on this exclusive use of the normal distribution, and tracks information about the type of distribution of each random variable and state properties, so as to prove the model-guide support match for realistic Pyro programs.

7.3 Analysis for Guide Parameter Differentiability

The second instance analysis aims at proving the differentiability of the density of a guide D_θ with respect to its parameters θ. It infers the continuous partial differentiability of multiple functions with respect to variables in the input state. The analysis is defined by the below abstraction:

$$\mathcal{T}^\sharp \triangleq \mathcal{P}(\mathit{Var}) \times \mathcal{P}(\mathit{Var}) \times \mathcal{P}(\mathit{Var} \times \mathit{Var}), \qquad \bot^\sharp \triangleq (\mathit{Var},\ \mathit{Var},\ \mathit{Var} \times \mathit{Var}),$$
$$\gamma(X, Y, R) \triangleq \Big\{ g \in \mathcal{D} \;\Big|\; \big(\forall x \in X.\, \forall s, r.\ g(s,r) \neq \bot \implies s(x) = (g(s,r))_s(x)\big)$$
$$\wedge \big(\forall y \in Y.\, \forall s, r.\ g(s,r) \neq \bot \implies \lambda v \in \mathbb{R}.\, \mathit{dens}(g, s[y \mapsto v])(r) \text{ is } C^1\big)$$
$$\wedge \big(\forall (z,u) \in R.\, \forall s, r.\ g(s,r) \neq \bot \implies \lambda v \in \mathbb{R}.\, (g(s[z \mapsto v], r))_s(u) \text{ is } \mathbb{R}\text{-valued and } C^1\big) \Big\}.$$

By "C^1", we mean that the relevant function is continuously differentiable. Being a $\mathbb{R}$-valued function in the last part requires that $g(s[z \mapsto v], r)$ be never $\bot$. An (X, Y, R) in $\mathcal{T}^\sharp$ expresses a property of a transformer $g \in \mathcal{D}$ (which can be viewed as semantic command) that g does not change variables in X, its density is C^1 with respect to each variable in Y in the input state, and for each $(z,u) \in R$, it assigns a real value to u in the output state in a C^1 manner with respect to z in the input state.

We now define the abstract operations induced by this abstraction. Given an expression E, we let $\mathcal{V}(E)$ denote the set of variables that occur in E, and we write $C^1(E)$ for the set of variables with

respect to which $[\![E]\!]$ is C^1 (based on classical differentiability rules). The definitions below follow from general principles such as the multivariate chain rule and account for discontinuities induced by conditions which break differentiability.

$$\begin{aligned}
\mathit{widen}((X_0,Y_0,R_0),(X_1,Y_1,R_1)) &= (X_0\cap X_1, Y_0\cap Y_1, R_0\cap R_1)\\
\mathit{cond}(E)^\sharp((X_0,Y_0,R_0),(X_1,Y_1,R_1)) &= (X_0\cap X_1, (Y_0\cap Y_1)\setminus \mathcal{V}(E),\\
&\qquad \{(z,u)\in R_0\cap R_1 \mid z\notin \mathcal{V}(E)\vee u\in X_0\cap X_1\})\\
\circ^\sharp((X_0,Y_0,R_0),(X_1,Y_1,R_1)) &= (X_0\cap X_1, \{x\in Y_1 \mid \forall y\in \mathit{Var}.\ (x,y)\in R_1\wedge y\in Y_0\},\\
&\qquad \{(z,v)\mid \forall u\in \mathit{Var}.\ (z,u)\in R_1\wedge (u,v)\in R_0\})\\
\mathit{skip}^\sharp &= (\mathit{Var},\mathit{Var},\mathit{Var}\times\mathit{Var})\\
\mathit{update}(x,E)^\sharp &= (\mathit{Var}\setminus\{x\},\mathit{Var},\mathit{Var}\times(\mathit{Var}\setminus\{x\})\cup\{(y,x)\mid y\in C^1(E)\})\\
\mathit{sample}(x,S,E_1,E_2)^\sharp &= (\mathit{Var}\setminus\{x\}, C^1(E_1)\setminus(\mathcal{V}(S)\cup\mathcal{V}(E_2)),\\
&\qquad (\mathit{Var}\setminus(\mathcal{V}(S)\cup\mathcal{V}(E_2)))\times\mathit{Var})\\
\mathit{score}(E_0,E_1,E_2)^\sharp &= (\mathit{Var}, (C^1(E_0)\cap C^1(E_1))\setminus\mathcal{V}(E_2), (\mathit{Var}\setminus\mathcal{V}(E_2))\times\mathit{Var})
\end{aligned}$$

To discharge the differentiability requirement R2, we need to run this analysis on a guide D_θ. If the Y component of the analysis result contains all the parameters θ (i.e., there exists (X,Y,R) such that $[\![D_\theta]\!]^\sharp = (X,Y,R)$ and $\theta\subseteq Y$), then the requirement R2 is met.

7.4 Analysis for Condition (10)

The third analysis extends the second by tracking and checking more properties. Its aim is to prove the condition (10). Just like the second analysis, it infers information about the continuous partial differentiability of multiple functions involving the output state and the density. Also, it checks whether the density of a given command C_0 has the form

$$\mathit{dens}([\![C_0]\!]_d, s)(r) = \mathit{dens}(g,s)(r) = \sum_{i=1}^{M} \mathbb{1}_{[r\in A_i]} \prod_{\alpha\in K_i} \mathcal{N}\big(r(\alpha); \mu_{(i,\alpha)}(s), \sigma_{(i,\alpha)}(s)\big) \tag{11}$$

for some *finite* M, and some A_i, K_i, $\mu_{(i,\alpha)}$ and $\sigma_{(i,\alpha)}$, and if so, it tracks properties of the $\mu_{(i,\alpha)}$ and $\sigma_{(i,\alpha)}$. Here is the abstraction for the analysis:

$$\begin{aligned}
&\mathcal{T}^\sharp \triangleq \{\top^\sharp\}\cup\big(\mathcal{P}(\mathit{Var})\times\mathcal{P}(\mathit{Var})\times\mathcal{P}(\mathit{Var}\times\mathit{Var})\big),\quad \bot^\sharp\triangleq(\mathit{Var},\ \mathit{Var},\ \mathit{Var}\times\mathit{Var}),\quad \gamma(\top^\sharp)\triangleq\mathcal{D},\\
&\gamma(X,Y,R)\triangleq\Big\{g\in\mathcal{D}\ \Big|\ g \text{ has the form (11)} \wedge \big(\forall x\in X.\forall s,r.\ g(s,r)\neq\bot \implies s(x)=(g(s,r))_s(x)\big)\\
&\quad\wedge\big(\forall s,r.\ g(s,r)\neq\bot\implies \lambda v\in\mathbb{R}.\,\mu_{(i,\alpha)}(s[\theta_j\mapsto v]) \text{ is } C^1 \text{ for all } i,j,\alpha\big)\\
&\quad\wedge\big(\forall s,r.\ g(s,r)\neq\bot\implies \lambda v\in\mathbb{R}.\,\sigma_{(i,\alpha)}(s[\theta_j\mapsto v]) \text{ is } C^1 \text{ for all } i,j,\alpha\big)\\
&\quad\wedge\big(\forall y\in Y.\forall s,r.\ g(s,r)\neq\bot\implies \lambda v\in\mathbb{R}.\,\mu_{(i,\alpha)}(s[y\mapsto v]) \text{ is } C^1 \text{ for all } i,\alpha\big)\\
&\quad\wedge\big(\forall y\in Y.\forall s,r.\ g(s,r)\neq\bot\implies \lambda v\in\mathbb{R}.\,\sigma_{(i,\alpha)}(s[y\mapsto v]) \text{ is } C^1 \text{ for all } i,\alpha\big)\\
&\quad\wedge\big(\forall(z,u)\in R.\forall s,r.\ g(s,r)\neq\bot\implies \lambda v\in\mathbb{R}.\,(g(s[z\mapsto v],r))_s(u) \text{ is } \mathbb{R}\text{-valued and } C^1\big)\Big\}.
\end{aligned}$$

The abstract operations are similar to those for the differentiability analysis thus we omit their definitions. We can use the analysis to prove the condition (10). We just need to run it on a guide D_θ and check whether $[\![D_\theta]\!]^\sharp$ is not $\top^\sharp$. If so, the condition holds.

7.5 Analysis for Condition (9)

The last instance is a static analysis that aims at proving the condition (9). The analysis checks whether a given command C_0 has a density of the following form:

$$\begin{aligned}
&\mathit{dens}(C_0,s)(r) = \mathit{dens}(g,s)(r) =\\
&\sum_{i=1}^{M} \mathbb{1}_{[r\in A_i]} \left(\prod_{\alpha\in K_i} \mathcal{N}\big(r(\alpha); \mu'_{(i,\alpha)}(s,r), \sigma'_{(i,\alpha)}(s,r)\big)\right)\left(\prod_{j\in[N_i]} \mathcal{N}\big(c_{(i,j)}; \mu''_{(i,j)}(s,r), \sigma''_{(i,j)}(s,r)\big)\right)
\end{aligned} \tag{12}$$

for some *finite* M, and some A_i, K_i, N_i, $\mu'_{(i,\alpha)}$, $\sigma'_{(i,\alpha)}$, $\mu''_{(i,j)}$, and $\sigma''_{(i,j)}$. If so, it tracks whether the $\mu'_{(i,\alpha)}$, $\sigma'_{(i,\alpha)}$, $\mu''_{(i,j)}$, and $\sigma''_{(i,j)}$ and some other functions can be bounded by affine exponentials on the input variables. The abstraction for the analysis is as follows:

$$\mathcal{T}^\sharp \triangleq \{\top^\sharp\} \cup \mathcal{P}(Var) \times \mathcal{P}(Var) \times (Var \rightharpoonup \mathcal{P}(Var)), \quad \bot^\sharp \triangleq (Var, Var, \lambda x{\in}Var.\, \emptyset), \quad \gamma(\top^\sharp) \triangleq \mathcal{D},$$

$$\begin{aligned}
\gamma(X,Y,R) \triangleq \big\{ g \in \mathcal{D} \;\big|\; & g \text{ has the form (12)} \wedge \big(\forall x \in X.\, \forall s, r.\; g(s,r) \neq \bot \Longrightarrow s(x) = (g(s,r))_s(x)\big) \\
& \wedge \big(\forall i \in [M], \alpha \in K_i.\, \exists \text{ an affine function } l \text{ from } [Y \cup K_i \to \mathbb{R}] \text{ to } \mathbb{R} \text{ such that for all } s, r, \\
& \quad g(s,r) \neq \bot \Longrightarrow \max(|\mu'_{(i,\alpha)}(s,r)|, \sigma'_{(i,\alpha)}(s,r), \sigma'_{(i,\alpha)}(s,r)^{-1}) \leq \exp(l(\{|s(x)|\}_{x\in Y}, \{|r(\beta)|\}_{\beta\in K_i}))\big) \\
& \wedge \big(\forall i \in [M], j \in [N_i].\, \exists \text{ an affine function } l \text{ from } [Y \cup K_i \to \mathbb{R}] \text{ to } \mathbb{R} \text{ such that for all } s, r, \\
& \quad g(s,r) \neq \bot \Longrightarrow \max(|\mu''_{(i,j)}(s,r)|, \sigma''_{(i,j)}(s,r), \sigma''_{(i,j)}(s,r)^{-1}) \leq \exp(l(\{|s(x)|\}_{x\in Y}, \{|r(\beta)|\}_{\beta\in K_i}))\big) \\
& \wedge \big(\forall y \in \mathrm{dom}(R), i \in [M].\, \exists \text{ an affine function } l \text{ from } [R(y) \cup K_i \to \mathbb{R}] \text{ to } \mathbb{R} \text{ such that for all } s, r, \\
& \quad g(s,r) \neq \bot \Longrightarrow (g(s,r))_s(y) \leq \exp(l(\{|s(x)|\}_{x\in R(y)}, \{|r(\beta)|\}_{\beta\in K_i}))\big)\big\}.
\end{aligned}$$

Here we use the notation $\{|s(x)|\}_{x\in Y}$ to mean a partial map from variables x in Y to values $|s(x)|$. The abstract operations that derive from this abstraction are quite similar to those of the differentiability analysis, therefore we do not detail them. To verify the condition (9), we run the analysis on a model C and check whether $[\![C]\!]^\sharp \neq \top^\sharp$. If the check succeeds, the condition holds.

8 STATIC ANALYSIS FOR MODEL-GUIDE PAIRS IN PYRO AND ITS EVALUATION

We present a static analysis that can verify the support correspondence for Pyro model-guide pairs. The analysis extends the first instance of the framework presented in §7. Our presentation focuses on the most salient aspects of this analysis and experimental results.

8.1 Some Features of Pyro Programs

Pyro is a probabilistic programming framework based on Python and PyTorch. It supports a wide spectrum of distributions and neural networks, and features commands for sampling and scoring. It comes with multiple inference engines, including SVI. We chose Pyro over other probabilistic programming languages (e.g., Anglican) because unlike most of other languages, in Pyro, SVI algorithms are considered as main inference engines and neural networks can be used together with probabilistic programming, which leads to more interesting examples.

In Pyro programs, random-variable names are often created at runtime, for instance by concatenating a fixed string with a sequence of dynamically-generated indices.

Example 8.1 (Use of indexed random-variable names). The code excerpt (of a model or guide program in Pyro) below samples $N \times M$ instances of independent random variables, and names them with "x_1_1", ..., "x_N_M".

```
for i in range(1,N+1):
  for j in range(1,M+1):
    val = pyro.sample("x_{}_{}".format(i,j), Normal(m,d))
```

□

Since Pyro is based on PyTorch and is adapted to implement data-science applications, Pyro programs heavily use multidimensional arrays, called tensors, and operations over them from PyTorch, in particular, element-wise operations and broadcasting. More precisely, when a binary operation (such as addition) is applied to two tensors of identical size, it outputs a new tensor of the same size where each output element is computed separately from the corresponding input elements. This principle often makes it possible to parallelise computations. Broadcasting occurs when a binary operation is applied to two tensors of different dimensions that can somehow be unified. Intuitively, it introduces and duplicates dimensions to produce tensors of identical dimensions, so that binary operations can be performed element-wise.

Tensors are also heavily used for sampling, which has several consequences. First, it means that an analysis targeted at Pyro programs should be able to track information about at least the dimensions of sampled tensors. Second, the dimensions of sampled tensors are grouped such that each of these groups has a different property with respect to probabilistic independence of tensor components. Pyro inference engines exploit this property for optimisation, for instance, via subsampling, but for this optimisation to be valid, the grouping of dimensions in a model should match that of a guide. Our analysis tracks information about dimension grouping of sampled tensors and the use of the Pyro construct called plate, which enables the optimisation just mentioned.

8.2 Abstract Domain

We extend our analysis from §7 so that it tracks not just how each part of a given program transforms states, but also which states can reach at each program point. We need the latter to get information about random variables with dynamically generated names that is precise enough for our verification task. To describe states, we rely on an abstract domain that consists of multiple subdomains (combined by product). Among them, the key part is $RDB^\sharp \triangleq [Str \rightharpoonup_{fin} \{\top\} \cup (\mathcal{P}_{fin}(Zone^\sharp) \times Dist^\sharp)]$, where $\rightharpoonup_{fin}$ denotes a finite partial map. $Dist^\sharp$ is an abstract domain whose element represents a set of elementary probability distributions, such as the standard normal distribution. The other $Zone^\sharp$ is our custom domain for zones, which express higher-dimensional rectangles in $\mathbb{N}^n$.

An element $r^\sharp \in RDB^\sharp$ means a set of concrete random databases r for each concrete store s. The domain of r consists of names that are obtained by concatenating baseline strings α in $\mathrm{dom}(r^\sharp)$ with index sequences. If $r^\sharp$ maps a baseline string α to $\top$, it does not give any information about index sequence and a probability distribution used for the random variable. Otherwise, the second component of $r^\sharp$ specifies the information about the distribution, and the first component a region of $\mathbb{N}^n$ that contains the index sequences used in the names. The region is described by a finite subset $\{Z_1^\sharp, \ldots, Z_n^\sharp\}$ of $Zone^\sharp$, which means the disjoint union of the rectangles represented by the $Z_i^\sharp$. To emphasise this union interpretation, we write $Z_1^\sharp \cup \ldots \cup Z_n^\sharp$ for the subset. The following table summarises the definitions of our abstract domain, based on $Zone^\sharp$:

$$
\begin{aligned}
RDB^\sharp &\triangleq [Str \rightharpoonup_{fin} \{\top\} \cup (\mathcal{P}_{fin}(Zone^\sharp) \times Dist^\sharp)];\\
Z^\sharp &\triangleq I_1^\sharp \times \ldots \times I_m^\sharp, \ \text{product of intervals in } \mathbb{N};\\
I^\sharp &\triangleq [B_l^\sharp, B_r^\sharp], \ \text{closed interval specified by bounds;}\\
B^\sharp &\triangleq c \mid x + c \mid x + c = c', \ \text{equality to constant, variable plus constant, or both.}
\end{aligned}
$$

A higher-dimensional rectangular zone $Z^\sharp$ is described by a finite sequence of intervals $I^\sharp$, each of which is made of two bounds. A bound $B^\sharp$ may be defined as one or two constraints which express that this bound is equal to a constant, or to a variable plus a constant, or both. This intuitive denotation defines a concretisation function $\gamma_r : RDB^\sharp \to \mathcal{P}(State)$.

Example 8.2. Consider the code of Example 8.1. After $i-1$ iterations in the main loop and j iterations in the last execution of the inner loop, we expect the analysis to come up with the invariant, $[\text{“}x\text{”} \mapsto ([1, i{-}1] \times [1, M] \cup [i, i] \times [1, j],\ \mathrm{normal}(m, d))]$. In turn, at the exit of the main loop, we expect the analysis to infer the invariant, $[\text{“}x\text{”} \mapsto ([1, i{=}N] \times [1, j{=}M],\ \mathrm{normal}(m, d))]$. □

In addition to these constraints over random databases, our analyser also uses an abstraction that maintains typing information and numerical constraints over variables. We do not fully formalise these constraints as the overall structure of the domain relies on a classical reduced product.

Finally, we describe the combination of the above state abstraction with the abstraction of §7.2. More precisely, we start with the abstract domain exposed in §7.2 (which we denote by $\mathcal{T}^\sharp$) and

build a novel abstract domain $\mathcal{T}_s^\sharp$ that also satisfies the requirements of §7.1. We let the set of abstract elements be $\mathcal{T}_s^\sharp \triangleq [RDB^\sharp \to \mathcal{T}^\sharp \times RDB^\sharp]$. Intuitively, such an element maps an input abstract random database into a set of functions together with an over-approximation of their output, when applied to this abstract input. The concretisation below formalises this: for all $t_s \in \mathcal{T}_s^\sharp$,

$$\gamma_s(t_s) \triangleq \Big\{ g \in \mathcal{D} \;\Big|\; \forall r_i^\sharp \in RDB^\sharp.\ t_s(r_i^\sharp) = (t, r_o^\sharp) \Longrightarrow \exists g' \in \gamma(t).\ \forall (s,r) \in \gamma_r(r_i^\sharp).\ g(s,r) = g'(s,r) \wedge \big(g(s,r) = \bot \vee (g(s,r)_s, g(s,r)_r) \in \gamma_r(r_o^\sharp)\big) \Big\}.$$

8.3 Computation of Loop Invariants

Although our static analysis for Pyro requires a state abstraction, its principles and structure are similar to those of the general analysis shown in §7. In the following, we first describe the integration of the state abstraction in the analysis of §7.2.

The abstract operations in $\mathcal{T}_s^\sharp$ can all be derived by lifting those in $\mathcal{T}^\sharp$ into functions. We illustrate this for the abstract composition $\circ_s^\sharp$ for $\mathcal{T}_s^\sharp$. Recall the operator $\circ^\sharp$ for $\mathcal{T}^\sharp$ in §7.2. Given $t_s, t_s' \in \mathcal{T}_s^\sharp$,

$$t_s \circ_s^\sharp t_s' \triangleq \lambda r_0^\sharp.\, (t \circ^\sharp t', r_2^\sharp) \quad \text{where } (t', r_1^\sharp) = t_s'(r_0^\sharp) \text{ and } (t, r_2^\sharp) = t_s(r_1^\sharp).$$

The other abstract operators over $\mathcal{T}_s^\sharp$ are defined in a similar manner.

As in most static analysis problems, the computation of precise loop invariants requires a carefully designed widening operator. In the case of zones, the analysis needs to generalise information about the bounds. We assume $r_0^\sharp, r_1^\sharp \in RDB^\sharp$ are abstract random databases, and present the main steps in the computation of the widening $\mathit{widen}(r_0^\sharp, r_1^\sharp)$.

(i) For each $r_i^\sharp$, we fuse zones $Z_0^\sharp, Z_1^\sharp$ in $r_i^\sharp$ into a single zone when $Z_0^\sharp$ and $Z_1^\sharp$ are the same except for one component and they represent adjacent high-dimensional rectangles. The following rewriting illustrates this step.

$$\big(\text{“}x\text{”} \mapsto \big([1, i{-}1] \times [1, j{=}M] \cup [i, i] \times [1, j{=}M],\ \mathrm{normal}(m, d)\big)\big) \rightsquigarrow \big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, j{=}M],\ \mathrm{normal}(m, d)\big)\big)$$

(ii) We generalise the intervals of corresponding zones in $r_0^\sharp$ and $r_1^\sharp$ using a weak version of unification on their bounds. The bounds of corresponding intervals in two such zones survive this step in a weakened form if they share at least one syntactically equal expression. Otherwise, the bounds are dropped, which causes the introduction of $\top$ into abstract random databases. The following rewriting instances illustrate this bound generalisation.

$$\mathit{widen}\big(\big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, j{-}1{=}M],\ \mathrm{normal}(m, d)\big)\big),\ \big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, j{=}M],\ \mathrm{normal}(m, d)\big)\big)\big) = \big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, M],\ \mathrm{normal}(m, d)\big)\big)$$

$$\mathit{widen}\big(\big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, j{-}1],\ \mathrm{normal}(m, d)\big)\big),\ \big(\text{“}x\text{”} \mapsto \big([1, i] \times [1, j{=}M],\ \mathrm{normal}(m, d)\big)\big)\big) = \big(\text{“}x\text{”} \mapsto \top\big)$$

The analysis also applies standard widening techniques to typing information and numerical constraints mentioned in §8.2. Finally, a model-guide pair can be verified if and only if their analyses return equivalent abstract random databases, without any name mapped into $\top$.

8.4 Experimental Evaluation

We have implemented a prototype analyser and carried out experiments so as to assess the effectiveness of our analysis to verify the support correspondence for Pyro model-guide pairs.[13] More precisely, we evaluated the following three research questions:

[13] Code is available at https://github.com/wonyeol/static-analysis-for-support-match.

(i) Can our analysis discover incorrect model-guide pairs in realistic probabilistic programs?
(ii) Can the analysis verify correct model-guide pairs despite the complexity of the definition?
(iii) Is the analysis efficient enough so that it can cope with realistic probabilistic programs?

Benchmark programs. We took example programs from the Pyro webpage that can be handled by standard SVI engines or can be naturally rewritten to be so. Those engines do not use Pyro's recent vectorised enumeration-based optimisation (enabled by the option `TraceEnum_ELBO`), and are run with the option `Trace_ELBO`. We made this choice because the optimisation is newly introduced and whether it is used or not changes the set of random variables used in a Pyro program.

We have applied our analysis to two groups of Pyro programs. The first is the *Pyro regression test suite* [Uber AI Labs 2019b], which comprises 66 examples exercising the basic programming patterns expected in Pyro programs. While each of these is small, they cover many standard patterns of both correct and incorrect model-guide pairs. Among these, 33 test cases use only `TraceEnum_ELBO` and fall outside the scope of our analysis, and 6 of the remaining 33 test cases come with two subsampling options. Hence, we can consider 39 experiments based on this regression suite.

The second is a collection of *examples* from the Pyro webpage [Uber AI Labs 2019a]. The webpage features 18 *Pyro examples*, among which 9 involve model-guide pairs (other examples use automatic guide generation, or do not perform SVI at all). Out of these 9 examples, 5 use `Trace_ELBO`, and three can be converted naturally into `Trace_ELBO` versions although they do not use `Trace_ELBO`. Thus, 8 Pyro examples fall within the scope of our analysis. These examples correspond to advanced probabilistic models[14] from the machine-learning literature, most of which use sophisticated neural networks and probabilistic modelling. Table 1 describes the structure of these 8 examples in detail.

Prototype analyser and results. Our analyser is implemented in OCaml, and supports the main data-structures and operations defined by Python, PyTorch, and Pyro. In particular, it precisely abstracts the shape of PyTorch tensor objects, the shape transformation information of PyTorch neural-network-related objects, the automatic broadcasting information of Pyro plate objects, and the shape of allocated indices for sample names, using the zone abstraction described above. It also supports common Pyro probability distributions, and can precisely cope with standard Pyro and PyTorch operations manipulating the Pyro distribution objects and PyTorch tensor objects. While our prototype supports a wide range of Python, PyTorch, and Pyro features, we point out that we did not implement a static analysis for the full Python (plus PyTorch and Pyro) language (e.g., no support for classes, dictionaries, named tuples, and user-defined functions).

The analysis results are summarised in Table 2 and are discussed in detail in the following. Run-times were measured on an Intel Core i7-7700 machine running Linux Ubuntu 16.04.

Discovery of invalid model-guide pairs. The analysis rejected two Pyro examples, `br` and `lda`, as incorrect due to an invalid model-guide pair. `br` is the Bayesian regression example discussed in §2.

For `br`, the analysis discovers that a random variable `sigma` is sampled from `Uniform(0.,10.)` in the model C, but from `Normal(..)` in the guide D_θ (Figure 2(a)). Since the support of `sigma` in D_θ is not a subset of that in C (i.e., $\mathbb{R} \not\subseteq [0, 10]$), the requirement R1 is violated. Thus, the SVI objective, $\mathrm{KL}(D_\theta \| C)$, is undefined, and `br` has an invalid model-guide pair.

For `lda`, the analysis discovers that a random variable `doc_topics` is sampled from `Dirichlet(..)` in the model C, but from `Delta(..)` in the guide D_θ. Since the reference measures of `doc_topics` in C and D_θ are different (the Lebesgue measure vs. the counting measure), $\mathrm{KL}(D_\theta \| C)$ cannot be computed by (4). For this reason, `lda` has an invalid model-guide pair. Our analyser tracks the

[14]The models include variational autoencoder (VAE) [Kingma and Welling 2014], semi-supervised VAE [Kingma et al. 2014], deep exponential family [Ranganath et al. 2015], attend-infer-repeat [Eslami et al. 2016], deep Markov model [Krishnan et al. 2017], inference compilation [Le et al. 2017], and amortised latent Dirichlet allocation [Srivastava and Sutton 2017].

Table 1. Key features of the model-guide pairs from Pyro examples. LoC denotes the lines of code of model and guide. The columns "Total #" show the number of objects/commands of each type used in model and guide, and the columns "Total dimension" show the total dimension of tensors in model and guide, either sampled from `sample` or used inside `score`, as well as the dimension of θ in guide.

			Total #				Total dimension		
Name	Corresponding probabilistic model	LoC	`for`	`plate`	`sample`	`score`	`sample`	`score`	θ
`br`	Bayesian regression	27	0	1	10	1	10	170	9
`csis`	Inference compilation	31	0	0	2	2	2	2	480
`lda`	Amortised latent Dirichlet allocation	76	0	5	8	1	21008	64000	121400
`vae`	Variational autoencoder (VAE)	91	0	2	2	1	25600	200704	353600
`sgdef`	Deep exponential family	94	0	8	12	1	231280	1310720	231280
`dmm`	Deep Markov model	246	3	2	2	1	640000	281600	594000
`ssvae`	Semi-supervised VAE	349	0	2	4	1	24000	156800	844000
`air`	Attend-infer-repeat	410	2	2	6	1	20736	160000	6040859

Table 2. Analysis results on two benchmark sets: Pyro test suite (Left) and Pyro examples (Right). The model-guide pairs from Pyro test suite are grouped into 7 categories, based on which type of plate objects are used. #Same (or #Diff) denotes the number of model-guide pairs for which the output of our analyser, valid or invalid, is the same as (or different from) the documented output. #Crash denotes the number of pairs for which our analyser crashes. The column "Time" shows the analysis time in seconds; on Left, it is averaged over those model-guide pairs (in each category) for which our analyser does not crash. The column "Valid?" shows the output of our analysis, valid or invalid.

Category	#Same	#Diff	#Crash	Time
No plates	9	0	0	0.001
Single `for`-plate	4	0	3	0.004
Nested `for`-plates	2	0	2	0.026
Single `with`-plate	5	0	0	0.001
Nested `with`-plates	7	2	0	0.002
Non-nested `with`-plates	2	0	0	0.002
Nested `for`-plate & `with`-plate	0	0	3	N/A
Total	29	2	8	0.003

Name	Valid?	Time
`br`	x	0.006
`csis`	o	0.007
`lda`	x	0.014
`vae`	o	0.005
`sgdef`	o	0.070
`dmm`	o	0.536
`ssvae`	o	0.013
`air`	o	4.093

reference measure implicitly by regarding the support of any distribution with Lebesgue measure, as disjoint from that of any distribution with counting measure (which is sound due to the aforementioned reason), and this allowed us to discover the issue of `lda`.

In both cases, the found correctness issues are unknown before and subtle. In particular, it turned out that `lda`, though incorrect when viewed as an example of SVI, is a valid implementation, because it performs not SVI but variational expectation-maximisation (EM) [Neal and Hinton 1998].[15] The `lda` uses an SVI engine just to solve an optimisation problem in variational EM (not to do SVI), and uses the `Delta` distribution to perform a particular realisation of the M-step in variational EM.

Verification of probabilistic programs relying on model-guide pairs. Among the Pyro test suite, the analysis successfully verifies 31 examples among 39. Interestingly, two of these 31 successful validations, highlighted in Table 2(a), correspond to cases that were flagged as "invalid model-guide pairs" in the Pyro git repository. Upon inspection, these two examples turn out to be correct.

[15] We thank an anonymous reviewer and Eli Bingham for pointing this out and explaining it in detail.

On the other hand, 8 examples from the Pyro test suite could not be verified due to the crashes of the analyser. One of these failures is due to the need to reason more precisely about the content of a `for` loop (e.g., using some partitioning techniques), and seven are due to the use of plates with subsampling, as ranges for `for` loops. Therefore, these failures could be resolved using existing static analysis techniques and a more precise handling of the semantics of Python constructions.

Moreover, 6 Pyro examples (among the 8 that we considered) were verified successfully, which means all correct Pyro examples were verified. Finally, we corrected the two examples that were rejected due to invalid model-guide pairs, and these two examples were also successfully verified.

Analysis efficiency. The analysis returned within a second on each program in the Pyro test suite, and on most of the Pyro examples. The slowest analysis was observed on `air`, which was analysed within 5 seconds. Most of the Pyro examples sample from and score with distributions of very high dimension arranged in complex tensors, using nested `for` and `plate`'s. While they are not large, they present a high degree of logical complexity, that is representative of realistic probabilistic programs. The fact that such programs get analysed within seconds shows that the analysis and the underlying abstract domain to describe zones, sampled dimensions, and distributions can generalise predicates quickly so that precise loop invariants can be computed.

9 RELATED WORK AND LIMITATION

Related work. As far as we know, the idea of using SVI for probabilistic programs first appeared in [Wingate and Weber 2013]. When further insights into how to create generic (sometimes also called black-box) SVI engines were found [Kucukelbir et al. 2015, 2017; Ranganath et al. 2014], the idea was tried for realistic probabilistic programming languages, such as Stan [Kucukelbir et al. 2015] and Anglican [van de Meent et al. 2016], resulting in impressive case studies [Kucukelbir et al. 2015]. The major inference engines for deep probabilistic programming languages [Bingham et al. 2019; Siddharth et al. 2017; Tran et al. 2018, 2016] are all based on SVI nowadays. However, we do not know of any prior work that attempts to reveal implicit assumptions made by these SVI engines and to discharge these assumptions manually or automatically, as we did in this paper.

The correctness of a different type of inference engines based on Monte-Carlo methods has been the subject of study in the PL community. Such engines have clearly formulated correctness conditions from Markov chain theory [Geyer 2011; Green 1995; Hastings 1970; Metropolis et al. 1953], such as ergodicity and correct stationarity. Tools from formal semantics have been employed to show that the inference engines satisfy these conditions [Borgström et al. 2016; Hur et al. 2015; Kiselyov 2016; Scibior et al. 2018]. While looking at different algorithms, some of these works consider more expressive languages than what we used in the paper, in particular, those supporting higher-order functions. One interesting direction is to extend our results to such expressive languages using the ideas from these works, especially the operational technique in [Borgström et al. 2016] and the denotational monad-based technique in [Scibior et al. 2018].

The consequence of having random choice in a programming language has been actively investigated by the semantics researchers from the early days [Borgström et al. 2016; Ehrhard et al. 2014; Heunen et al. 2017; Jones and Plotkin 1989; Kozen 1981; Smolka et al. 2017; Staton 2017; Staton et al. 2016; Toronto et al. 2015; Vákár et al. 2019]. Our work uses the technique developed in this endeavour, such as Giry monad and denotational formulation of idealised importance sampling [Staton et al. 2016]. Also, just as we connected the measure semantics with the density semantics, [Kozen 1981] and [Borgström et al. 2016] related two semantics with similar flavours, although the considered languages (with or without continuous distribution and score statements) and the style of semantics (operational vs denotational) are different. Our density semantics uses a reference measure built out of Lebesgue measure as in [Bhat et al. 2012, 2013; Hur et al. 2015]. This choice

is found to cause an issue when the score statement is treated not as a scoring mechanism but in terms of conditional expectation [Wu et al. 2018]. How to resolve this matter is still open.

Static analyses for probabilistic programming languages or languages with probabilistic choice typically attempt to check probabilistic properties [Chakarov and Sankaranarayanan 2013; Cousot and Monerau 2012; Monniaux 2000; Wang et al. 2018], or perform posterior inference [Chaganty et al. 2013; Wang et al. 2018], or find information useful for posterior inference [Nori et al. 2014]. For instance, [Monniaux 2000] defines a probabilistic abstract interpretation framework, which is applied to estimate the failure probability [Monniaux 2001]. More recently, [Cousot and Monerau 2012] sets up a general framework to design probabilistic abstract interpretations, which lift conventional static analysis tools to a probabilistic setup, and [Wang et al. 2018] proposes a framework for analysing probabilistic programs based on hyper-graphs with probabilistic actions on edges. Our program analyses aim at a different type of verification tasks, namely, proving the safety requirements imposed by the SVI engines. Static analyses for checking the continuity properties of programs are proposed in [Chaudhuri et al. 2010]. Some techniques used in those analyses may help track the kind of smoothness properties considered in this paper precisely.

Limitation. We described four static analyses (§7.2-§7.5) for discharging the requirements R1-R6, and developed one of the four analyses further (§8.2-§8.4) to build a prototype analyser for actual Pyro programs. Here we clarify the underlying assumptions and limitations of these analyses.

The static analysis for model-guide support match (§7.2) was implemented into our static analyser for Pyro (§8.4) but with additional extensions (§8.2-§8.3), so that it does not make the assumption in §6.1; the assumption was introduced mainly to develop analyses for the requirements R3-R6. Hence, our static analyser handles both continuous and discrete random variables.

On the other hand, other analyses (§7.3-§7.5) are unimplemented and make the assumption in §6.1. We expect that the assumption can be relaxed, without much difficulty, to permit continuous or discrete distributions having finite entropy and moments of all degrees, because our proofs of theorems and lemmas bound various expectations mostly by entropies and moments. It would be more challenging, however, to relax the assumption to allow distributions not having finite entropy and moments of all degrees, or models having unboundedly many random variables (of any kind). In particular, addressing the later limitation might require techniques developed for proving that probabilistic systems has finite expected execution time.

This paper focuses on a particular optimisation objective (5) for SVI, but we point out that other objectives have been proposed for different inference or learning algorithms, such as variational EM (e.g., `lda` in §8.4) and importance weighted autoencoder [Burda et al. 2016]. One interesting research direction is to develop techniques for verifying the well-definedness of these objectives.

ACKNOWLEDGMENTS

We thank Fritz Obermeyer and Eli Bingham for explaining the subtleties of Pyro and suggesting us to try the Pyro regression test suite. Sam Staton, Ohad Kammar and Matthijs Vákár helped us to understand the semantics of recursion in the probabilistic programming languages better. Lee, Yang and Yu were supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177).

REFERENCES

Sooraj Bhat, Ashish Agarwal, Richard W. Vuduc, and Alexander G. Gray. 2012. A type theory for probability density functions. In *Principles of Programming Languages (POPL)*. 545–556.

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 508–522.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *International Conference on Functional Programming (ICFP)*. 33–46.

Yuri Burda, Roger B. Grosse, and Ruslan Salakhutdinov. 2016. Importance Weighted Autoencoders. In *International Conference on Learning Representations (ICLR)*.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32.

Arun Tejasvi Chaganty, Aditya V. Nori, and Sriram K. Rajamani. 2013. Efficiently Sampling Probabilistic Programs via Program Analysis. In *Artificial Intelligence and Statistics (AISTATS)*. 153–160.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification (CAV)*. 511–526.

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Principles of Programming Languages (POPL)*. 57–70.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL)*. 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*. 269–282.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2, 4 (1992), 511–547.

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *European Symposium on Programming (ESOP)*. 169–193.

Thomas Ehrhard, Christine Tasson, and Michele Pagani. 2014. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In *Principles of Programming Languages (POPL)*. 309–320.

S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. 2016. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. In *Neural Information Processing Systems (NIPS)*. 3233–3241.

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV)*. 62–83.

Charles J. Geyer. 2011. Introduction to Markov Chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*, Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng (Eds.). Chapman and Hall/CRC, Chapter 1, 3–48.

Hamid Ghourchian, Amin Gohari, and Arash Amini. 2017. Existence and Continuity of Differential Entropy for a Class of Distributions. *IEEE Communications Letters* 21, 7 (2017), 1469–1472.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*. 220–229.

Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2014. Tabular: A Schema-driven Probabilistic Programming Language. In *Principles of Programming Languages (POPL)*. 321–334.

Peter J. Green. 1995. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika* 82, 4 (1995), 711–732.

Wilfred Keith Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Logic in Computer Science (LICS)*. 1–12.

Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. 2013. Stochastic Variational Inference. *Journal of Machine Learning Research* 14 (2013), 1303–1347.

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2015. A Provably Correct Sampler for Probabilistic Programs. In *Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*. 475–488.

C. Jones and Gordon D. Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Logic in Computer Science (LICS)*. 186–195.

Diederik P. Kingma, Danilo J. Rezende, Shakir Mohamed, and Max Welling. 2014. Semi-supervised Learning with Deep Generative Models. In *Neural Information Processing Systems (NIPS)*. 3581–3589.

Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations (ICLR)*.

Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 357–376.

Achim Klenke. 2014. *Probability Theory: A Comprehensive Course* (second ed.). Springer-Verlag London.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350.

Rahul G. Krishnan, Uri Shalit, and David Sontag. 2017. Structured Inference Networks for Nonlinear State Space Models. In *AAAI Conference on Artificial Intelligence (AAAI)*. 2101–2109.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. Automatic Variational Inference in Stan. In *Neural Information Processing Systems (NIPS)*. 568–576.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2017. Automatic Differentiation Variational Inference. *Journal of Machine Learning Research* 18 (2017), 14:1–14:45.

Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *Artificial Intelligence and Statistics (AISTATS)*. 1338–1348.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards Verified Stochastic Variational Inference for Probabilistic Programs. *arXiv:1907.08827* (2019).

Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv:1404.0099* (2014).

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Matthew Mirman, Timon Gehr, and Martin T. Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning (ICML)*. 3575–3583.

David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *Static Analysis Symposium (SAS)*. 322–339.

David Monniaux. 2001. Backwards Abstract Interpretation of Probabilistic Programs. In *European Symposium on Programming (ESOP)*. 367–382.

Chandra Nair, Balaji Prabhakar, and Devavrat Shah. 2006. On Entropy for Mixtures of Discrete and Continuous Variables. *arXiv:cs/0607075* (2006).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming (FLOPS)*. 62–79.

Radford M. Neal and Geoffrey E. Hinton. 1998. A View of the Em Algorithm that Justifies Incremental, Sparse, and other Variants. In *Learning in Graphical Models*. 355–368.

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. 2476–2482.

John William Paisley, David M. Blei, and Michael I. Jordan. 2012. Variational Bayesian Inference with Stochastic Search. In *International Conference on Machine Learning (ICML)*. 1363–1370.

Rajesh Ranganath, Sean Gerrish, and David M. Blei. 2014. Black Box Variational Inference. In *Artificial Intelligence and Statistics (AISTATS)*. 814–822.

Rajesh Ranganath, Linpeng Tang, Laurent Charlin, and David Blei. 2015. Deep Exponential Families. In *Artificial Intelligence and Statistics (AISTATS)*. 762–771.

Adam Scibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational validation of higher-order Bayesian inference. *PACMPL* 2, POPL (2018), 60:1–60:29.

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. 2017. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. In *Neural Information Processing Systems (NIPS)*. 5927–5937.

Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor meets scott: semantic foundations for probabilistic networks. In *Principles of Programming Languages (POPL)*. 557–571.

Akash Srivastava and Charles A. Sutton. 2017. Autoencoding Variational Inference For Topic Models. In *International Conference on Learning Representations (ICLR)*.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming (ESOP)*. 855–879.

Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Logic in Computer Science (LICS)*. 525–534.

Neil Toronto, Jay McCarthy, and David Van Horn. 2015. Running Probabilistic Programs Backwards. In *European Symposium on Programming (ESOP)*. 53–79.

Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, and Alexey Radul. 2018. Simple, Distributed, and Accelerated Probabilistic Programming. In *Neural Information Processing Systems (NeurIPS)*. 7609–7620.

Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja R. Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv:1610.09787* (2016).

Uber AI Labs. 2019a. Pyro examples. http://pyro.ai/examples/. Version used: April 1, 2019.

Uber AI Labs. 2019b. Pyro regression test suite. https://github.com/pyro-ppl/pyro/blob/dev/tests/infer/test_valid_models.py. Version used: March 1, 2019.

Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *PACMPL* 3, POPL (2019), 36:1–36:29.

Jan-Willem van de Meent, Brooks Paige, David Tolpin, and Frank D. Wood. 2016. Black-Box Policy Search with Probabilistic Programs. In *Artificial Intelligence and Statistics (AISTATS)*. 1195–1204.

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *Programming Language Design and Implementation (PLDI)*. 513–528.

Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8, 3-4 (1992), 229–256.

David Wingate and Theophane Weber. 2013. Automated Variational Inference in Probabilistic Programming. *arXiv:1301.1299* (2013).

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics (AISTATS)*. 1024–1032.

Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart J. Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *International Conference on Machine Learning (ICML)*. 5339–5348.

Hongseok Yang. 2019. Implementing Inference Algorithms for Probabilistic Programs. https://github.com/hongseok-yang/probprog19/blob/master/Lectures/Lecture6/Note6.pdf. Lecture Note of the 2019 Course on Probabilistic Programming at KAIST.