Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages

ALEXANDER K. LEW, Massachusetts Institute of Technology
MARCO F. CUSUMANO-TOWNER, Massachusetts Institute of Technology
BENJAMIN SHERMAN, Massachusetts Institute of Technology
MICHAEL CARBIN, Massachusetts Institute of Technology
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology

Modern probabilistic programming languages aim to formalize and automate key aspects of probabilistic modeling and inference. Many languages provide constructs for *programmable inference* that enable developers to improve inference speed and accuracy by tailoring an algorithm for use with a particular model or dataset. Unfortunately, it is easy to use these constructs to write unsound programs that appear to run correctly but produce incorrect results. To address this problem, we present a denotational semantics for programmable inference in higher-order probabilistic programming languages, along with a type system that ensures that well-typed inference programs are sound by construction. A central insight is that the type of a probabilistic expression can track the space of its possible *execution traces*, not just the type of value that it returns, as these traces are often the objects that inference algorithms manipulate. We use our semantics and type system to establish soundness properties of custom inference programs that use constructs for variational, sequential Monte Carlo, importance sampling, and Markov chain Monte Carlo inference.

CCS Concepts: • **Mathematics of computing** → *Probabilistic inference problems*; *Variational methods*; *Metropolis-Hastings algorithm*; *Sequential Monte Carlo methods*; • **Theory of computation** → *Semantics and reasoning*; *Denotational semantics*; • **Software and its engineering** → *Formal language definitions.*

Additional Key Words and Phrases: Probabilistic programming, type systems, programmable inference

ACM Reference Format:
Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2020. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. *Proc. ACM Program. Lang.* 4, POPL, Article 19 (January 2020), 31 pages. https://doi.org/10.1145/3371087

1 INTRODUCTION

Probabilistic modeling and inference are central tools in multiple fields, including artificial intelligence, statistics, and robotics, but can be difficult for practitioners to apply correctly [Gelman et al. 2013; Russell and Norvig 2016; Thrun et al. 2005]. To make these tools more accessible, researchers have developed probabilistic programming languages that automate aspects of modeling and inference [Carpenter et al. 2017; Goodman et al. 2008; Ścibior et al. 2015; Wood et al. 2014], and more recently, formal semantics for these languages, enabling the validation of their inference engines [Heunen et al. 2017; Ścibior et al. 2017; Staton et al. 2016]. For many problems, however, the generic inference algorithms used in most languages converge too slowly [Mansinghka et al. 2018]. To improve the speed and accuracy of inference, several newer languages feature constructs for *programmable inference* [Bingham et al. 2019; Cusumano-Towner et al. 2019; Ge et al. 2018; Mansinghka et al. 2018, 2014; Murray 2013; Zinkov and Shan 2016], which let developers tailor an

2475-1421/2020/1-ART19
https://doi.org/10.1145/3371087

inference algorithm for use with a particular model or dataset. These features have been shown to enable state-of-the-art performance on problems such as automatic data modeling [Saad et al. 2019], 3D human body pose estimation [Cusumano-Towner et al. 2019], and optimal experiment design [Foster et al. 2019].

In this paper, we propose a new denotational approach for understanding probabilistic programming languages, such as Venture [Mansinghka et al. 2018, 2014], Pyro [Bingham et al. 2019] and Gen [Cusumano-Towner et al. 2019], that leverage program tracing to support programmable inference constructs. These constructs are expressive but not always safe: using them, it is possible to write unsound inference code that crashes the inference engine unpredictably, or that runs to completion but yields meaningless or biased results. To address this challenge, we use our semantics to develop and validate sound-by-construction versions of a variety of programmable inference features. Key to our approach is our type system, in which a probabilistic program's type precisely characterizes the space of its possible execution traces. We use these "trace types" to enforce measure-theoretic conditions relevant to the soundness of inference, such as the *absolute continuity* of one program's distribution over execution traces with respect to another's. This lets us ensure key soundness properties for a broad class of user-written inference programs, including custom proposals for importance sampling, sequential Monte Carlo, and Markov chain Monte Carlo, as well as custom approximating families for variational inference.

1.1 Contributions

We identify four core contributions of this work:

(1) **Trace types for probabilistic programs.** We introduce a type system for probabilistic programs in which a program's type precisely characterizes the space of its possible execution traces (Section 5). The type system handles programs from an expressive probabilistic language, with higher-order functions, discrete and continuous random choices, support-altering branches, and stochastic while loops.
(2) **Inference with sound-by-construction custom proposals and variational families**. Trace types enable us to extend our core language with sound-by-construction inference programming features. We introduce custom-proposal variants of importance sampling, sequential Monte Carlo, and MCMC (Sections 6.1, 6.2, 6.3), along with typing rules, denotational semantics, and a soundness theorem for each. We also introduce constructs for variational inference with custom variational families (Section 6.4), and typing rules that enforce certain preconditions for their soundness.
(3) **Combinators for sound composition of MCMC kernels**. We introduce a set of combinators enabling custom compositions of MCMC kernels (including conditional branching, sequencing, repeating, and random mixture), and use trace types to track the class of models for which such a composite kernel is stationary (Section 6.3).
(4) **Validation of correctness within a unified denotational framework.** We extend the framework of Ścibior et al. [2017] to provide a unified treatment of many programmable inference constructs: we model custom scheduling for MCMC (from Turing [Ge et al. 2018] and Venture [Mansinghka et al. 2014]), custom proposals for Monte Carlo algorithms (from Gen [Cusumano-Towner et al. 2019] and Pyro [Bingham et al. 2019]), and custom variational families for variational inference (from Gen, Pyro and Edward [Tran et al. 2017]).

To demonstrate that our approach is practical, we embedded our language and type system for models and auxiliary distributions (proposals, kernels, and variational families) into Haskell. We found that the (valid) probabilistic programs in this paper all type-check in the Haskell implementation, and even have their trace types automatically inferred by GHC.

2 BACKGROUND

2.1 Modeling with probabilistic programs

In probabilistic programming languages, users encode generative models by writing programs that generate random latent variables and simulate observations. We can interpret these simulators as defining probability distributions over their possible *execution traces*, records of all random variables sampled in the course of a particular execution. Consider the toy program p in Figure 1, adapted from Pyro's tutorials, which models the weighing of a small object of unknown mass. The model encodes a probability distribution on two related random variables, each introduced with a labeled **sample** statement: *weight* represents the unknown mass (in grams, perhaps) of some small object, and *measurement* represents the object's observed weight as reported by an unreliable scale. We model our uncertain beliefs about the object's weight, prior to obtaining a measurement, by sampling it from a **gamma** distribution, and the noisy weighing process itself using **normal**. A typical execution trace might be $\{weight = 1.03, measurement = 1.42\}$.

2.2 Sound and unsound programmable inference

We may now wish to infer a likely *weight* given an observed *measurement* (say, of 0.5 grams). To do so, we can invoke an algorithm for Bayesian inference, with model p and query $\{measurement = 0.5\}$. Probabilistic programming languages typically come packaged with implementations of popular Monte Carlo and variational inference algorithms, which can be used to estimate conditional expectations or to obtain samples of *weight* approximately distributed according to the posterior.

But Monte Carlo and variational inference algorithms are typically parameterized not only by a model and query, but also by some *auxiliary* probability distribution, the choice of which can have a great impact on an inference algorithm's rate of convergence or accuracy, in ways that also depend on the particular details of the model and query. In importance sampling and sequential Monte Carlo, we must specify *proposal* distributions; in Markov chain Monte Carlo (MCMC), *transition kernels*; and in variational inference, *variational families*. Much research in probabilistic programming has focused on finding generic ways of constructing proposals, MCMC transition kernels, and variational families suitable for use with any model and dataset. Although these generic proposals make the inference engine's interface simpler for users (provide a model, then choose from a menu of algorithms), they also limit the system's applicability: for many problems, generic algorithms cannot reliably give accurate results in a reasonable amount of time.

Programmable inference constructs [Bingham et al. 2019; Cusumano-Towner et al. 2019; Ge et al. 2018; Mansinghka et al. 2018, 2014; Murray 2013; Zinkov and Shan 2016] give users control over the creative, high-level task of specifying proposals, kernels, and variational families, while automating the calculations necessary to implement the underlying algorithm, such as the computation of Metropolis-Hastings acceptance probabilities in MCMC or the optimization of a variational family's parameters. Specifying custom proposals, transition kernels, or variational families for inference can help Monte Carlo algorithms to converge more rapidly or improve the accuracy of variational algorithms. Unfortunately, wielding this power can be dangerous: using programmable inference constructs, one can write unsound inference programs that yield incorrect results. We now describe three broad classes of inference algorithms, and the pitfalls that users must avoid when using their programmable variants (Figure 1). Our goal in the remainder of the paper will then be to formalize these programmable inference algorithms and develop tools for ensuring that they are used soundly.

2.2.1 Importance sampling (IS) and sequential Monte Carlo (SMC). IS and SMC approximate an intractable target distribution (in Bayesian inference, the posterior) by sampling from a tractable *proposal* distribution, then assigning a score to each sample to correct for the discrepancy. Estimates

Model and Query

$$p = \textbf{do}\,\{\, w \leftarrow \textbf{sample}_{weight}\,(\textbf{gamma}\ 2\ 1)$$
$$\qquad\qquad \textbf{sample}_{measurement}\,(\textbf{normal}\ w\ 0.2)\,\}$$
$$obs = \{measurement = 0.5\}$$

Importance Sampling (Unsound Proposal)

$$q_1 = \textbf{sample}_{weight}\ \textbf{uniform}$$
$$\textbf{importance}\ p\ obs\ q_1$$

Importance Sampling (Sound Proposal)

$$q_1' = \textbf{sample}_{weight}\ (\textbf{gamma}\ 2\ 0.25)$$
$$\textbf{importance}\ p\ obs\ q_1'$$

MCMC (Unsound Kernel)

$$k_1 = \textbf{mh}\ (\lambda\, t.\ \textbf{sample}_{weight}\ (\textbf{normal}_+\ t.weight\ 0.2))$$
$$k_2 = \textbf{mh}\ (\lambda\, t.\ \textbf{sample}_{weight}\ (\textbf{normal}_+\ t.weight\ 1.0))$$
$$k = \textbf{seq}_{\mathcal{K}}\ (\textbf{if}_{\mathcal{K}}\ (\lambda t.\ t.weight \leq 2)\ k_1)$$
$$\qquad (\textbf{if}_{\mathcal{K}}\ (\lambda t.\ t.weight > 2)\ k_2)$$
$$\textbf{applyKernel}\ (p.observe\ obs)\ k\ 100$$

MCMC (Sound Kernel)

$$k' = \textbf{mh}\ (\lambda t.\ \textbf{sample}_{weight}$$
$$\qquad (\textbf{normal}_+\ t.weight$$
$$\qquad\quad (\textbf{if}\ (t.weight \leq 2)$$
$$\qquad\qquad \textbf{then}\ 0.2\ \textbf{else}\ 1.0)))$$
$$\textbf{applyKernel}\ (p.observe\ obs)\ k'\ 100$$

Variational Inference (Unsound Approximation)

$$q_2 = \lambda\ (a, b).\ \textbf{sample}_{weight}\ (\textbf{normal}\ a\ b)$$
$$\textbf{svi}\ p\ obs\ q_2\ (1, 1)$$

Variational Inference (Sound Approximation)

$$q_2' = \lambda\ (a, b).\ \textbf{sample}_{weight}\ (\textbf{normal}_+\ a\ b)$$
$$\textbf{svi}\ p\ obs\ q_2'\ (1, 1)$$

Fig. 1. A model p, a query *obs*, and three unsound (left) and sound (right) inference programs implementing importance sampling (IS), Markov chain Monte Carlo (MCMC), and variational inference (VI) algorithms respectively. Our type system rejects the unsound programs statically.

of expectations can then be calculated by taking weighted averages. For any proposal that satisfies mild technical conditions, these algorithms converge asymptotically to the desired target distribution as more samples are taken, but convergence may be prohibitively slow if the proposal is chosen poorly; for many problems, the generic proposals used in languages without programmable inference [Goodman et al. 2008] fail to give accurate results within a reasonable amount of time.

Programmable inference features let users specify custom proposals, which may speed up convergence, in the same probabilistic language that they use to write models (Figure 1, q_1 and q_1'). But for the algorithm to converge at all, users must be careful to ensure that the model and proposal programs are compatible: the model's posterior distribution over execution traces must be *absolutely continuous* with respect to the proposal's. In other words, any set of traces with non-zero probability under the posterior distribution must also have non-zero probability under the proposal. In Figure 1, q_1 is *not* a valid proposal distribution for the *weight*-estimation problem: it will only ever produce samples on the interval $(0, 1)$. No matter how many samples we take, importance sampling with q_1 cannot possibly approximate the true posterior, which places non-zero probability on the event that *weight* is larger than 1 gram (even though the observed measurement is 0.5). The sound proposal q_1' samples instead from a **gamma** distribution, which has support on all of $\mathbb{R}_+ = \{r \in \mathbb{R} \mid r > 0\}$.

2.2.2 Markov chain Monte Carlo (MCMC). MCMC algorithms sample from an intractable target distribution by repeatedly applying a stochastic *transition kernel* to an initial sample, designed

to ensure that the chain eventually converges to the desired target distribution (i.e., the target is the stationary distribution, or fixed point, of the transition kernel). The generic transition kernels [Wingate et al. 2011] used in some languages have the right stationary distribution, but may take prohibitively long to converge.

With programmable inference, users gain two degrees of freedom in constructing their own transition kernels. First, they can construct arbitrary Metropolis-Hastings transition kernels (Figure 1, k_1, k_2, and k'), by writing probabilistic code implementing a transition from a current trace t to a proposed new trace. The **mh** construct extends the proposal with an *accept-reject step* that flips a weighted coin to decide whether to move to a proposed next state; the weight of the coin is automatically computed so as to ensure that the resulting kernel is stationary for the desired target distribution. The kernels k_1 and k_2 are *Gaussian drift* kernels: they propose a new *weight* by sampling from a truncated normal distribution centered at the current value of *weight*.

Second, users can combine simple transition kernels into larger ones via sequencing, repetition, probabilistic mixture, and conditional execution, all of which preserve stationarity under mild conditions (Figure 1, k). But users must ensure that these conditions are met, or the resulting Markov chain may target the wrong distribution or fail to converge at all. The kernel k does not meet these conditions: it uses $\mathbf{if}_{\mathcal{K}}$ to decide which of two sub-kernels to execute, but these sub-kernels may end up negating the $\mathbf{if}_{\mathcal{K}}$ condition. As a result, the kernel k no longer satisfies detailed balance, and targets the wrong distribution. The kernel k' implements the same logic, but branches *inside* the **mh** proposal, so the automatically computed accept/reject probabilities can account for the asymmetry.

2.2.3 Variational inference (VI). Variational inference [Zhang et al. 2019] uses optimization (e.g., stochastic gradient descent) to find parameters θ that bring a parametric family of distributions q_θ (e.g., a neural network) close to the desired target distribution. Unlike Monte Carlo algorithms, variational inference does not necessarily converge to the desired target distribution, even in the limit of infinite computation. Programmable versions of variational inference allow users to specify hand-designed variational families (Figure 1, q_2 and q_2'), which may be better able to approximate the target distribution than the simple, general-purpose variational families used by some probabilistic programming languages without programmable inference [Carpenter et al. 2017; Ge et al. 2018; Wingate and Weber 2013]. In order for the optimization problem to be well-posed, however, the user must ensure that the KL divergence between the true posterior and members of the variational family is finite.

The unsound variational family q_2, parameterized by a mean a and standard deviation b, represents a normal distribution over the *weight* variable. The **svi** operation, provided by the language backend, optimizes the parameters of q_2 to minimize the divergence from this variational approximation to the true posterior on *weight*. We might hope that once this optimization is complete, a would represent our best guess at the object's true weight, and b our remaining uncertainty, having seen only a single measurement (0.5). But the divergence that **svi** is supposedly minimizing is in this case infinite, no matter the parameters a and b. This is because the support of the **gamma** distribution, and thus the *weight* random variable, is $\mathbb{R}_+ = \{r \in \mathbb{R} \mid r > 0\}$, whereas the **normal** distribution used in q has support on the entire real line. In practice, running the inference program with q_2 may result in division by zero (or taking the logarithm of zero), producing nan or crashing a system many iterations deep into an optimization loop. Even if optimization runs to completion, it is unclear how we should interpret the resulting "best-fit" parameters, since the objective function for the optimization is everywhere equal to ∞. The program q_2' implements a sound variational approximation that fixes the issue by sampling instead from the truncated normal distribution $\mathbf{normal}_+\ a\ b$.

For labels $l \in \mathcal{L}$, finite label sets $I \subset \mathcal{L}$, natural numbers $n \in \mathbb{N}$, variables x, y, constant terms c:

Deterministic core

$$\begin{aligned}
\text{Types } \tau, \sigma ::= {} & \mathbf{1} \mid \mathbf{U} \mid \mathbf{R}_+ \mid \mathbf{R} \mid \overline{\mathbf{R}}_{\geq 0} \mid \mathbf{N} \mid \mathbf{Fin}\, n \mid \mathbf{Vec}\, n\, \tau \mid \mathbf{List}\, \tau \mid \\
& \tau + \sigma \mid \tau \times \sigma \mid \tau \to \sigma \mid \textstyle\prod_{i \in I} \tau_i \\
\text{Terms } t, s, u, v ::= {} & x \mid c \mid () \mid \mathbf{inl}_\sigma\, t \mid \mathbf{inr}_\tau\, t \mid \mathbf{match}\, t\, \mathbf{with}\, \mathbf{inl}\, x \to s\,, \mathbf{inr}\, y \to u \mid (t, s) \mid \\
& \lambda x.\, t \mid t\, s \mid [t_1, \ldots, t_n] \mid \{\,\} \mid \{l = t\} \mid t +\!\!+ s \mid t.l \mid \mathbf{restrict}_\tau\, t
\end{aligned}$$

Probabilistic programming with trace types

$$\begin{aligned}
\text{Types } \tau_p ::= {} & \mathcal{D}\, \tau \text{ (probability distributions)} \mid \mathcal{U}\, \tau \text{ (conditioned distributions)} \mid \\
& \mathcal{P}\, \tau\, \sigma \text{ (probabilistic programs)} \\
\text{Distribution terms } t_d, s_d ::= {} & \mathbf{uniform} \mid \mathbf{bernoulli}\, t \mid \mathbf{categorical}_n\, t \mid \mathbf{normal}\, t\, s \mid \\
& \mathbf{gamma}\, t\, s \mid \mathbf{beta}\, t\, s \mid \mathbf{geometric}\, t \mid t_p.\mathit{traced} \mid t_p.\mathit{observe}\, s \\
\text{Program terms } t_p, s_p ::= {} & \mathbf{sample}_l\, t_d \mid \mathbf{withProbability}_l\, t\, s_p\, \mathbf{else}\, u_p \mid \\
& \mathbf{for}_l\, x\, \mathbf{inRandomRange}\, t_d\, s_p \mid \mathbf{foreach}_l\, x\, \mathbf{in}\, t\, s_p \mid \\
& \mathbf{while}_l\, (x := t;\ \mathit{min}(s, u))\, v_p \mid \mathcal{P}.\mathbf{return}\, t \mid \mathcal{P}.\mathbf{do}\, \{x \leftarrow t_p;\ s_p\} \mid \\
& \mathcal{P}.\mathbf{do}\, \{t_p;\ s_p\} \mid \mathcal{P}.\mathbf{do}\, \{\mathbf{let}\, x = t;\ s_p\}
\end{aligned}$$

Inference programming

$$\begin{aligned}
\text{Types } \tau_i ::= {} & \mathcal{M}\, \tau \text{ (measures / samplers)} \mid \mathcal{K}\, \tau\, \sigma \text{ (MCMC kernels)} \\
\text{Terms } t_i, s_i, u_i ::= {} & \mathbf{sample} \mid \mathbf{score}\, t \mid \mathcal{M}.\mathbf{return}\, t \mid \mathcal{M}.\mathbf{do}\, \{x \leftarrow t_i;\ s_i\} \mid \mathcal{M}.\mathbf{do}\, \{t_i; s_i\} \mid \\
& \mathcal{M}.\mathbf{do}\, \{\mathbf{let}\, x = t; s_i\} \mid t_p.\mathit{returnValue} \mid t_d.\mathit{sampler} \mid t_d.\mathit{density} \mid \\
& \mathbf{importance}\, t_p\, s\, u_p \mid \mathbf{particleFilter}\, t_p\, s_p\, u_p\, v \mid \mathbf{svi}\, t_p\, s\, u_p\, v \mid \\
& \mathbf{trainAmortized}\, t_p\, s_p\, u \mid \mathbf{mh}\, t_p \mid \mathbf{if}_\mathcal{K}\, t\, s_i \mid \mathbf{seq}_\mathcal{K}\, t_i\, s_i \mid \mathbf{mix}_\mathcal{K}\, t\, s_i\, u_i \mid \\
& \mathbf{repeat}_\mathcal{K}\, t\, s_i \mid \mathbf{applyKernel}\, t_d\, s_i\, u \mid \mathcal{B}[\![\tau]\!]
\end{aligned}$$

Fig. 2. Syntax of our core calculus. We extend the language developed by Ścibior et al. [2017] (white) with new constructs for probabilistic programming with trace types and for programmable inference (gray).

3 OVERVIEW OF OUR APPROACH

In this paper, we formalize a variety of programmable inference constructs, and mechanize the process of ensuring that they are used soundly.

3.1 Probabilistic programming with trace types

We begin by building a core calculus for probabilistic programming with trace types (Section 5). The calculus is designed to capture key features of modern probabilistic programming languages, such

as Gen and Pyro, that leverage program tracing to support programmable inference. Like them, our language supports higher-order functions, branching, and loops. But unlike those languages, which are dynamically typed, our core calculus features a static type system for probabilistic programs that tracks their possible execution traces. We use these types to enforce measure-theoretic conditions for the soundness of inference, enabling us to implement a library for sound-by-construction inference programming in Section 6.

Our work builds on the framework introduced by Ścibior et al. [2017], which we review in Section 4. In Section 5, we extend their simply typed calculus for measures and samplers (Figure 2, in white) with constructs for probabilistic programming with trace types (Figure 2, middle):

(1) ***Density-carrying probability distributions***, of type $\mathcal{D}\,\tau$, represent the primitive distributions from which probabilistic programs can sample. Functions for constructing widely used distributions, both discrete and continuous, are built in, including **normal**, **gamma**, **bernoulli**, **uniform**, and $\mathbf{categorical}_n$. We require that the types of these distributions be precise enough to capture their supports exactly: formally, each distribution of type $\mathcal{D}\,\tau$ must have a strictly positive density with respect to the stock base measure we assign to the type τ. For example, the distribution **normal** 0 1 has type $\mathcal{D}\,\mathbf{R}$, **gamma** 2 1 has type $\mathcal{D}\,\mathbf{R}_+$, and $\mathbf{categorical}_3\ [0.2,\ 0.3,\ 0.5]$ has type $\mathcal{D}\,(\mathbf{Fin}\,3)$.

(2) Users build more complex distributions by writing ***traced probabilistic programs***, which may encode either generative models or auxiliary programs for inference (that is, proposal distributions, transition kernels, and variational approximations). Traced probabilistic programs are constructed monadically using Haskell-inspired $\mathcal{P}.\mathbf{do}\ \{\ldots\}$ blocks. Within these blocks, programmers interleave *labeled* sample statements from primitive distributions, written $\mathbf{sample}_l\ d$, with deterministic computations, stochastic branches, and loops. Probabilistic programs are assigned types of the form $\mathcal{P}\,\tau\,\alpha$: the two parameters τ and α are the probabilistic program's *trace type* and *return type* respectively. The trace type τ is a record type that tracks the labels at which a probabilistic program samples, and the types of the values sampled at each label. We call it a trace type because records of type τ can be thought of as execution traces of the probabilistic program, recording every random value sampled during a particular execution. The data of a probabilistic program p includes a distribution over traces of type τ (which we refer to as $p.traced : \mathcal{D}\,\tau$), as well as a deterministic function from traces to return values ($p.returnValue : \tau \to \alpha$).

(3) These probabilistic programs do not allow **score** statements for conditioning. Following Gen, users write full generative models, specifying joint probability distributions over all variables of interest. Then, outside of the probabilistic code, the user can *condition* the model on observations. This is done using the syntax $p.observe\ t$, where t is an *observation trace*, a record specifying the values observed for some of the random choices taken by p. The result is a conditioned distribution ($\mathcal{U}\,\tau$, a potentially unnormalized density-carrying measure) on traces of unobserved variables. This design, which keeps **score** statements out of probabilistic programs, allows us to ensure that programs representing proposals and variational distributions express probability distributions that are easy to sample from, rather than unnormalized, conditioned distributions.

The key feature of this design is that probabilistic programs always sample at a statically known set of labels, captured in their types. This might seem at first glance to be overly restrictive. Gen and Pyro both support dynamically computed names for sample statements, which allows, for example, a loop that samples a random variable with a different name at each iteration. They also support the construction of probabilistic programs that *may* sample at a label, but not with probability 1. Our language also supports such programs, though our programming model is more restrictive. In

Trace types capture the names and domains of random variables, but not sampling order.

$\mathbf{do}\,\{\,\mathbf{sample}_x\ (\mathbf{normal}\ 0\ 1)$
$\quad\mathbf{sample}_z\ (\mathbf{geometric}\ 3)\,\}$
$\mathcal{P}\,\{x : \mathbf{R}, z : \mathbf{N}\}\,\mathbf{N}$

$\mathbf{do}\,\{\,\mathbf{sample}_z\ (\mathbf{poisson}\ 2)$
$\quad\mathbf{sample}_x\ (\mathbf{gamma}\ 1\ 1)\,\}$
$\mathcal{P}\,\{x : \mathbf{R}_+, z : \mathbf{N}\}\,\mathbf{R}_+$

$\mathbf{do}\,\{\,\mathbf{sample}_z\ (\mathbf{normal}\ 1\ 1)$
$\quad\mathbf{sample}_z\ (\mathbf{bernoulli}\ 0.2)\,\}$
ill-typed

Traces with sum types reflect stochastic control flow.

```
do {
  isBiased ← sample_b (bernoulli 0.1)
  p ← if isBiased then
    sample_p (beta 10 1)
  else sample_p uniform
  sample_coin (bernoulli p)
}
```
$\mathcal{P}\,\{b : \mathbf{B}, p : \mathbf{U}, \mathit{coin} : \mathbf{B}\}\,\mathbf{B}$

```
do {
  p ← withProbability_p 0.1 do {
    isLow ← sample_isLow (bernoulli 0.5)
    return (if isLow then 0.01 else 0.99)
  } else return 0.5
  sample_coin (bernoulli p)
}
```
$\mathcal{P}\,\{p : \{\mathit{isLow} : \mathbf{B}\} + \{\}, \mathit{coin} : \mathbf{B}\}\,\mathbf{B}$

Looping constructs introduce lists and vectors into program traces.

```
do {
  pts ← for_pts i inRandomRange (poisson 3) do {
    x ← sample_x (normal i 1)
    sample_y (normal x 1)
  }
  return (map (λy. 2 * y) pts)
}
```
$\mathcal{P}\,\{\mathit{pts} : \mathbf{List}\,\{x : \mathbf{R}, y : \mathbf{R}\}\}\,(\mathbf{List}\,\mathbf{R})$

```
do {
  pts ← foreach_pts x in [1, 2, 3] do {
    sample_y (normal x 1)
  }
  return (map (λy. 2 * y) (toList pts))
}
```
$\mathcal{P}\,\{\mathit{pts} : \mathbf{Vec}\,3\,\{y : \mathbf{R}\}\}\,(\mathbf{List}\,\mathbf{R})$

Fig. 3. Examples of traced probabilistic programs and their types

our language, control flow constructs for branching and looping are themselves annotated with a label l; the type associated with l in the program's trace reflects the possible control flow paths (Figure 3). Branches result in sum types in the trace, reflecting that the *subtrace* associated with the label l might have one of two possible structures. Loops result in lists of subtraces, reflecting that we may not know statically how many iterations a loop will run, but we do know the labels and types of each iteration's random variables. (When we *do* know the number of iterations statically, we use sized vectors instead of lists.)

3.2 Sound programmable inference

With trace types in hand, we can formulate sound-by-construction versions of a variety of programmable inference constructs.

Returning to the examples in Figure 1, we can now assign types to both the model p and probabilistic components of the inference programs, such as the variational families q_2 and q_2':

$p : \mathcal{P}\,\{weight : \mathbf{R}_+, measurement : \mathbf{R}\}\,\mathbf{R}$
$q_2 : \mathbf{R} \times \mathbf{R} \to \mathcal{P}\,\{weight : \mathbf{R}\}\,\mathbf{R}$
svi $p\,\{\ measurement = 0.5\ \}\ q_2$

$obs : \{\ measurement : \mathbf{R}\ \}$
$q_2' : \mathbf{R} \times \mathbf{R} \to \mathcal{P}\,\{weight : \mathbf{R}_+\}\,\mathbf{R}$
svi $p\,\{\ measurement = 0.5\ \}\ q_2'$

Note that the types assigned to the label *weight* differ in p and in q. The **svi** command has a typing rule that requires the record concatenation of the observation trace's type ($\{measurement : \mathbf{R}\}$) and the variational family's trace type ($\{weight : \mathbf{R}\}$) to equal the model's trace type ($\{weight : \mathbf{R}_+, measurement : \mathbf{R}\}$). In this case, there is a mismatch between the two, enabling us to reject the faulty inference command **svi** $p\,\{measurement = 0.5\}\,q_2$ statically.

Apart from the **svi** construct introduced above, we support several others, each accepting a model, expressed as a probabilistic program, as well as one or more additional probabilistic program(s) that parameterize the inference algorithm:

Importance sampling with a custom proposal. The **importance** construct (Section 6.1) produces an importance sampler given a model, observations, and a custom proposal distribution expressed as a probabilistic program. The type checker verifies that the model is *absolutely continuous* with respect to the proposal distribution, and that the query is compatible with the model. This guarantees that the importance sampling program is *properly weighted* [Liu and Chen 1998] with respect to the desired posterior.

Particle filtering with custom initialization and step proposals. The **particleFilter** construct (Section 6.2) produces a sequential Monte Carlo sampler given a model, a sequence of observed data, and initial and step proposal distributions expressed as probabilistic programs. The type system again ensures proper weighting by verifying that the model is compatible with each step's proposal distribution and observed data.

Constructing stationary Metropolis-Hastings kernels from custom multivariate proposals. The **mh** construct (Section 6.3) produces an MCMC kernel from a user-specified proposal distribution using the Metropolis-Hastings accept/reject rule, guaranteeing that the resulting kernel is provably stationary with respect to the target distribution. Because proposals are arbitrary (well-typed) probabilistic programs, they can propose jointly to many random choices at once, even in models with stochastic control flow.

Combinators for sound composition of MCMC kernels. Users can compose MCMC kernels into more complex kernels using the kernel combinators $\mathbf{seq}_{\mathcal{K}}$ (sequencing), $\mathbf{mix}_{\mathcal{K}}$ (random choice of kernel), $\mathbf{if}_{\mathcal{K}}$ (conditional application of kernel), and $\mathbf{repeat}_{\mathcal{K}}$, while provably retaining the correct stationary distribution (Section 6.3). The type system prevents common errors in composing kernels incorrectly, such as conditionally applying a kernel that negates the condition predicate.

Training an amortized inference program. The **trainAmortized** construct (Section 6.4) generates simulated data from a model, and uses it to train a custom variational family (which may, for example, invoke a neural network) to approximately match the posterior distribution, using densities computed in a provably sound manner. (Although amortized inference and SVI are typically implemented using automatic differentiation and gradient descent, we do not investigate the semantics of differentiation or differentiability in this work. Instead, for variational inference, we prove only certain preconditions for soundness.)

3.3 Programmer workflow

A user of our language will typically adhere to the following workflow:

(1) **Defining a model.** The user defines a model $p : \mathcal{P}\,(\tau +\!\!+ \sigma)\,\alpha$ as a probabilistic program. We write its trace type as the record concatenation (Section 4.1) of two smaller trace types, τ and σ, to emphasize that models generally contain a subset of random variables (those in σ) representing the quantities over which we would like to do inference, and a subset of random variables (those in τ) whose values we have already observed and which we wish to use for conditioning.
(2) **Encoding observations.** The user encodes a dataset on which to *condition* the model as a *query trace* t of type τ.
(3) **Default inference.** The user *could* write $(p.observe_\tau\ t).sampler$ to condition p on t and obtain the default-inference sampler targeting the posterior.
(4) **Customizing inference.** To achieve better performance, the user can instead choose to define an auxiliary program q, often of type $\mathcal{P}\,\sigma\,\mathbf{1}$, whose traces are the same shape as the posterior's, and whose code incorporates domain knowledge about the dataset or model that might improve the convergence rate of inference. The program q may encode a proposal distribution, transition kernel, or variational family.
(5) **Executing the algorithm.** The user invokes a programmable inference command to construct a custom sampler, e.g. **importance** $p\ t\ q$. If q is chosen well, this sampler may produce accurate posterior samples with less computation than the default algorithm. For each inference command (e.g. **importance**), we provide a typing rule, and a proof of relevant soundness properties (for example, that for well-typed importance sampling programs, the resulting sampler targets the desired posterior). (Note that although a user could theoretically construct the efficient sampler manually, doing so could require performing error-prone calculations of densities, gradients, and/or Metropolis-Hastings acceptance ratios manually, and the user would lose any guarantees that their samplers targeted the desired posterior. Programmable inference allows the user to construct efficient samplers at a high level of abstraction, while automating these low-level details.)
(6) **Composing with other algorithms.** Because the samplers produced by our programmable inference constructs are terms in the language of Ścibior et al. [2017], they are compatible with the more sophisticated compositions of inference algorithms presented in Ścibior et al. [2018]. For example, users could incorporate custom sound-by-construction transition kernels into an SMC sampler as rejuvenation moves, or use custom sound-by-construction SMC proposals to estimate model likelihoods in particle-marginal MH.

3.4 End-to-end example

To make this programming model more concrete, we now walk through an end-to-end example of modeling and inference programming within our core calculus (Figure 4).

3.4.1 A model for curve-fitting. Suppose we are trying to solve the following inference problem: given a dataset of ordered pairs $(x_i, y_i) \in \mathbb{R}^2$ in the plane (lines 1-2), we would like to infer the degree and coefficients of a polynomial $f(x)$ that characterizes the relationship between x and y. We begin with a probabilistic program (lines 3-11) that specifies a probability distribution over several unknown quantities (the degree, the coefficients, and a global noise level), representing our uncertain beliefs prior to seeing the data. We use a $\mathbf{for}_l \ldots$ **inRandomRange** loop to choose a random degree, and sample that many coefficients.

The type annotation on line 3 indicates that *prior* is a probabilistic program. Its output type is $(\mathbf{R} \to \mathbf{R}) \times \mathbf{R}_+$, because it returns a polynomial of type $\mathbf{R} \to \mathbf{R}$ and a noise level of type $\mathbf{R}_+$. More interesting is the trace type: it specifies that execution traces of this program are records, mapping the name *noise* to positive reals, and the name *coeffs* to lists of *subtraces*, each of which maps the

```
1  xs : Vec 7 R = [−3, −2, −1, 0, 1, 2, 3]
2  ys : Vec 7 {y : R} = [{y = 3.1}, {y = 1.8}, {y = 1.1}, {y = −0.2}, {y = −1.2}, {y = −2.1}, {y = −3.0}]

3  prior : P {noise : R+, coeffs : List {c : R}} ((R → R) × R+) = P.do {
4    z ← sample_noise (gamma 1 1)
5    terms ← for_coeffs i inRandomRange (geometric 0.4) do {
6      coeff ← sample_c (normal 0 1)
7      return λ x. coeff ∗ x^i
8    }
9    let f = λ x. fold (λ y t. y + t x) 0 terms
10   return (f, z)
11 }

12 p : P {noise : R+, coeffs : List {c : R}, data : Vec 7 {y : R}} (Vec 7 R) = P.do {
13   (f, z) ← prior
14   foreach_data x in xs do {
15     sample_y (normal (f x) z)
16   }
17 }

18 // A 384−parameter, two−layer neural network.
19 nn : Vec 384 R → Vec 7 (R × R) → {π : U, μ_c : R, σ_c : R+, logNoise : R}

20 q : (Vec 384 R × R+) → {data : Vec 7 {y : R}} → P {noise : R+, coeffs : List {c : R}} R+ =
21  λ (θ, σ) obs. P.do {
22    let ys = mapv (λ a. a.y) obs.data
23    let nnInputs = λ n resids. zipv (mapv (λ x. x^n) xs) resids
24    let initialState = {n = 0, residuals = ys, nnResults = nn θ (nnInputs 0 ys)}
25    finalState ← while_coeffs (s := initialState; min(s.nnResults.π, 0.99)) do {
26      c ← sample_c (normal s.nnResults.μ_c s.nnResults.σ_c)
27      let newResids = mapv (λ (x, y). y − c ∗ x^(s.n)) (zipv xs s.residuals)
28      let newN = s.n + 1
29      return {n = newN, residuals = newResids, nnResults = nn θ (nnInputs newN newResids)}
30    }
31    sample_noise (lognormal finalState.nnResults.logNoise σ)
32  }

33 doInference : M {coeffs : List R, noise : R+} = M.do {
34   (θ̂, σ̂) ← trainAmortized p q (θ_init, σ_init)
35   importance p {data = ys} (q (θ̂, σ̂) {data = ys})
36 }
```

Fig. 4. An end-to-end example: curve-fitting.

name c to a real number. The record does not explicitly store the polynomial's degree (sampled from the **geometric** distribution on line 5), but it is implicitly captured as the length of *coeffs*.

3.4.2 Looping over data. Having specified a prior, we now add on a likelihood (lines 12-17), modeling the process by which a dataset of (x, y) points is generated. We use $\mathbf{foreach}_I$ to loop over our vector of xs, sampling a y coordinate for each. The trace type (line 12) captures that traces of p contain a vector, *data*, of y values.

3.4.3 A neural network for inference. We now use the same probabilistic programming constructs to specify a custom *inference network*, for use with amortized inference or importance sampling. Our variational family q (lines 20-32) has the same trace type as *prior*, but uses a neural network (line 19), with our dataset as input, to parameterize the distributions from which it samples. The program q takes two inputs: a tuple of parameters, and an *observation trace* containing observed y values. Its job is to propose values stochastically for the unobserved random choices: the coefficients of the polynomial and the global noise level. To do this, it uses a $\mathbf{while}_I$ loop to repeatedly call a neural network to estimate the next coefficient in the polynomial. In order to generate coefficient i, the neural network accepts as input ordered pairs (x^i, r_i), where $r_i = y - \sum_{j=0}^{i-1} c_j x^j$ is the *residual* not yet explained by already-sampled coefficients. The neural network returns several values: a probability π that additional coefficients should still be sampled, a mean and standard deviation from which to sample the next coefficient (if it exists), and (if it doesn't exist) a best estimate of a global noise value that could explain the remaining residuals. On each iteration, the $\mathbf{while}_I$ loop on line 25 is entered with probability $\min(s.nnResults.\pi, 0.99)$.

Crucially, even though this program samples variables in a very different manner from the *prior*, with different control flow, its trace type is the same, reflecting that the two programs specify *mutually absolutely continuous* distributions over traces. This is why it's sound to use one as part of an inference algorithm for the other.

3.4.4 Performing inference. We perform inference (lines 33-37) in two steps: first, we obtain good values of the parameters θ and σ using **trainAmortized**. Then we use those parameters to perform importance sampling with q as a custom proposal, using the **importance** command. Both of these inference library commands have typing rules that ensure that p, q, and the observed data are compatible. In order to sequence the two inference commands, which both produce values of *sampler* type, we use monadic composition for samplers ($\mathcal{M}$.**do**).

4 PRELIMINARIES

In order to reason carefully about the correctness of programmable inference, we need a formal framework for understanding what mathematical objects probabilistic programs represent, and how inference algorithms operate on them. For this, we turn to the work of Ścibior et al. [2017], who introduced an elegant approach for reasoning formally about inference in probabilistic programs. They develop a simply typed language in which terms can be interpreted either as denoting mathematical measures, or as denoting samplers that target those measures: importance samplers, particle filters, and MCMC samplers are all obtained by interpreting the same terms in different ways. In this section, we review their development, so that we can extend it with trace types and programmable inference constructs in the sequel.

4.1 Deterministic core calculus

We begin by defining a deterministic core language (Figure 2, top), an extension to the simply-typed λ-calculus. It includes unit (**1**), product, and sum types, as well as the open unit interval (**U**), the reals (**R**), the positive reals ($\mathbf{R}_+$, does not include 0), the non-negative extended reals ($\overline{\mathbf{R}}_{\geq 0}$, includes

$$\frac{}{\Gamma \vdash \{\} : \{\}} \qquad \frac{\Gamma \vdash t : \tau \qquad l \in \mathcal{L}}{\Gamma \vdash \{l = t\} : \{l : \tau\}} \qquad \frac{\Gamma \vdash t : \prod_{i \in I} \tau_i \qquad \Gamma \vdash s : \prod_{j \in J} \tau_j \qquad I \cap J = \emptyset}{\Gamma \vdash t +\!\!+ s : \prod_{i \in I \cup J} \tau_i}$$

$$\frac{\Gamma \vdash t : \prod_{i \in I} \tau_i \qquad l \in I}{\Gamma \vdash p.l : \tau_l} \qquad \frac{\Gamma \vdash t : \prod_{i \in I \cup J} \tau_i \qquad \sigma = \prod_{i \in I} \tau_i \qquad I \cap J = \emptyset}{\Gamma \vdash \mathbf{restrict}_\sigma\, t : \sigma}$$

Fig. 5. Typing rules for records.

0 and ∞), the natural numbers (**N**), lists (**List** τ), length-indexed vectors (**Vec** n τ), and bounded natural numbers (**Fin** n). When it makes sense to, we include as constant terms the inclusion maps ι_σ^τ. For example, we have $\iota^{\mathbf{U}}_{\mathbf{R}_+}$, $\iota^{\mathbf{R}_+}_{\mathbf{R}}$, $\iota^{\mathbf{Vec}n\tau}_{\mathbf{List}\tau}$ for every n and τ, and so on.

Although our language does not support general recursion, we include as constant terms the standard **fold** operations on lists and natural numbers, as well as the list constructors $\mathbf{Nil}_\tau$ and $\mathbf{Cons}_\tau$. (Ścibior et al. [2017] also introduce user-defined iso-recursive types, but we omit that development here.) Vectors of the same length can be $\mathbf{zipv}_n^{\tau,\sigma}$'d together into vectors of tuples, and $\mathbf{mapv}_n^{\tau,\sigma}$ applies a function to each element of a vector. We use standard syntactic sugar, e.g., treating $\mathbf{1} + \mathbf{1} = \mathbf{B}$ as the type of Booleans, and using **if** ... **then** ... **else** for pattern matching on them. We write vector literals as $[t_1, \ldots, t_n]$.

As a minor extension to Ścibior et al. [2017], we consider in this paper a deterministic core that also contains record types (Figure 5). We first fix a countable set $\mathcal{L}$ of *labels*, as well as an enumeration of $\mathcal{L}$, inducing a total order on labels. (For example, we might set $\mathcal{L}$ to be the set of strings with lexicographic ordering.) Then the record type $\prod_{i \in I} \tau_i$ assigns to each label i in a finite set $I \subset \mathcal{L}$ a type τ_i. Given a term t of record type $\prod_{i \in I} \tau_i$, and a label $l \in I$, we can write $t.l$ to pick out the corresponding record element. If s is a term of another record type $\sigma = \prod_{j \in J} \tau_j$, and $I \cap J = \emptyset$, then we write $t +\!\!+ s$ for the record concatenation of t and s, yielding a value of type $\prod_{i \in I \cup J} \tau_i$. When, on the other hand, $J \subset I$, we can write $\mathbf{restrict}_\sigma t$, which deletes entries from the record t to obtain a smaller record of type σ. We write $\{l_1 : \tau_{l_1}, \ldots, l_n : \tau_{l_n}\}$ to mean the record type $\prod_{l \in I} \tau_l$; the desugaring of this expression does not depend on the order in which we write the labels. At the type level, we write $\tau +\!\!+ \sigma$ to mean the type containing the labels of τ and the labels of σ, where these two label sets are disjoint. When such a type appears in a premise of a deductive inference rule, as in Figure 10, the disjointness condition is implied as an additional premise.

It is straight-forward to give this language a semantics in the category **Set**, i.e., to interpret types as sets, variable contexts Γ as set products (of the sets denoted by all the types in the context), constant terms of type A as functions $[\![\mathbf{1}]\!] \to [\![A]\!]$ picking out certain elements, and well-typed terms-in-context $\Gamma \vdash t : A$ as functions $[\![\Gamma]\!] \to [\![A]\!]$ taking valuations of their free variables to elements of the sets denoted by their types. We take the base types (unit, the natural numbers, the reals, and so on) to represent their obvious mathematical counterparts. We can then follow the standard set-theoretic constructions for Cartesian products, co-products (disjoint unions), and exponentials (function types). As for record types, we take them to denote product spaces: a record type $\tau = \prod_{i \in I} \tau_i$ denotes the product of its component types' denotations, sorted by label.

4.2 Quasi-Borel spaces

We would like to extend our deterministic core with probabilistic constructs for building measures and samplers. But the set-theoretic semantics we described above are insufficient for this: sets must be augmented with some additional structure if we wish to talk rigorously about "probability distributions" or "measures" over them.

Measure theory. The standard approach for doing so comes from measure theory. We turn a set $|X|$ into a *measurable space* X by equipping it with a σ-algebra $\Sigma_X \subseteq \mathcal{P}(|X|)$, specifying its *measurable subsets*. Then a *measure* μ on X is just a function taking X's measurable subsets to non-negative real numbers (possibly ∞), such that every measure μ gives zero weight to the empty set ($\mu(\emptyset) = 0$), and the measure of the countable union of disjoint sets is the term-wise sum of their measures ($\mu(\cup X_i) = \sum \mu(X_i)$). When $\mu(|X|) = 1$, we say μ is a *probability measure*; we can think of the elements of Σ_X as being events, and μ as characterizing their probabilities. Whenever $\mu(|X|)$ is neither zero nor infinite, we can *normalize* μ to obtain a probability measure, by dividing by its normalizing constant $Z = \mu(|X|)$. Unnormalized measures are very common in practice, often arising, e.g., from distributions on groups of random variables that have been conditioned on observations of a subset of them.

For the ground types in our language, it is possible to equip σ-algebras that make them *standard Borel spaces*, measurable spaces isomorphic to either $\{1, \ldots, n\}$, $\mathbb{N}$, or $\mathbb{R}$. The set of standard Borel spaces is closed under countable products and coproducts, so sum types, product types, lists, records, and vectors built from our base types can all be interpreted as standard Borel spaces, too. But the category **Meas** of measurable spaces and measurable maps (functions $f : |X| \to |Y|$ such that when $A \in \Sigma_Y$, $f^{-1}(A) \in \Sigma_X$) is not cartesian-closed, which means that there is no good way to give meanings to function types $A \to B$ [Heunen et al. 2018].

Quasi-Borel spaces. These difficulties led Heunen et al. [2017] to develop the category **QBS** of *quasi-Borel spaces* and their morphisms. Like a measurable space, a quasi-Borel space X also consists of a carrier set $|X|$ along with some additional structure. But instead of axiomatizing its measurable subsets Σ_X, a quasi-Borel space axiomatizes its *admissible random elements* M_X. A random element of X is a function from $\mathbb{R}$ to $|X|$. The set M_X must satisfy certain closure properties, e.g., closure under precomposition with measurable $\mathbb{R} \to \mathbb{R}$ functions. The morphisms of **QBS** are functions $f : |X| \to |Y|$ such that $f \circ \alpha \in M_Y$ whenever $\alpha \in M_X$.

The category **QBS** is cartesian-closed, so can model languages with higher-order functions, and it also has a strong commutative monad of measures (which we come to in a moment). We can take our set-theoretic denotational semantics and extend it to one based in **QBS** by noting that **QBS** contains as objects all the standard Borel spaces X, equipped with the random elements $M_X = \mathbf{Meas}(\mathbb{R}, X)$, the measurable maps from $\mathbb{R}$ to X. Function types denote exponentials in **QBS**.

Following Ścibior et al. [2017], we define a *measure* over a quasi-Borel space $[\![\tau]\!]$ as a triple (Ω, α, μ) of a standard Borel space Ω, a morphism $\alpha : \Omega \to [\![\tau]\!]$, and a (measure-theoretic) σ-finite measure μ on Ω. We consider two such measures *equivalent* if they are indistinguishable as integrators: i.e., if for all morphisms $f : [\![\tau]\!] \to \overline{\mathbb{R}}_+$, $\int_{\Omega_1} f(\alpha_1(x))\mu_1(dx) = \int_{\Omega_2} f(\alpha_2(x))\mu_2(dx)$. We can now introduce, for each type τ, a type $\mathcal{M}\,\tau$; the elements of the space $[\![\mathcal{M}\,\tau]\!]$ are these equivalence classes of quasi-Borel measures. Ścibior et al. [2017] shows that the types $\mathcal{M}\,\tau$ form a strong commutative monad in **QBS**. We do not review the details of their construction here, but do offer some intuition about the meanings of the monadic **return** and bind in the next section.

$$\frac{}{\Gamma \vdash \mathbf{sample} : \mathcal{M}\,\mathbf{U}} \qquad \frac{\Gamma \vdash w : \overline{\mathbf{R}}_{\geq 0}}{\Gamma \vdash \mathbf{score}\ w : \mathcal{M}\,\mathbf{1}} \qquad \frac{\Gamma \vdash x : \alpha}{\Gamma \vdash \mathbf{return}\ x : \mathcal{M}\,\alpha}$$

$$\frac{\Gamma \vdash \mu : \mathcal{M}\,\alpha \qquad \Gamma, x : \alpha \vdash t : \mathcal{M}\,\beta}{\Gamma \vdash \mathcal{M}.\mathbf{do}\,\{x \leftarrow \mu; t\} : \mathcal{M}\,\beta}$$

Fig. 6. Monadic construction of measures and samplers.

4.3 Constructing measures and samplers

We now introduce constructs for building terms of measure type $\mathcal{M}\,\tau$ (Figure 6). Such terms can be viewed either as denoting quasi-Borel measures or as denoting samplers (i.e., implementations of inference algorithms). Although users of our proposed architecture may not work with them directly, the **sample**, **score**, **return**, and $\mathcal{M}$.**do** constructs we introduce in this section are the fundamental building blocks with which we will implement the sound-by-construction inference programming features we introduce later on.

Measure terms as quasi-Borel measures. Because quasi-Borel measures are equivalent up to the integration operators they define, we can understand terms of measure type as specifying integrators (and, when they are probability measures, expectation-takers): a measure μ on $[\![\tau]\!]$ maps functions $f : \tau \to \overline{\mathbf{R}}_{\geq 0}$ to their μ-integrals over the space $[\![\tau]\!]$. Taking this view, the **sample** primitive integrates a function $f : \mathbf{U} \to \overline{\mathbf{R}}_{\geq 0}$ over the unit interval (or, equivalently, takes its expectation with respect to a uniform random variable), and the expression **score** w yields an integrator that, given an $f : \mathbf{1} \to \overline{\mathbf{R}}_{\geq 0}$, returns $w * f()$.

The monadic **return** x is the Dirac measure centered at x, or as an integrator, $\lambda f.f(x)$: the expectation of a function f with respect to the Dirac distribution at x is just $f(x)$. The fact that $\mathcal{M}$ is a monad also gives us *bind* functions ${}_{\alpha}\gg\!=_{\beta}$ for all pairs of types α and β, and we interpret $\mathcal{M}$.**do**$\{x \leftarrow \mu; t\}$ as desugaring to $\mu\,{}_{\alpha}\gg\!=_{\beta}(\lambda x.t)$. As an integrator, $\mu_\alpha \gg\!=_\beta k$ is $\lambda f.\mu(\lambda x.k(x)(f))$, or, using integral notation, $\lambda f.\int_{[\![\alpha]\!]}\int_{[\![\beta]\!]} f(y)k(x)(dy)\mu(dx)$. (We find this $\lambda\, f.$ notation, which we borrow from Narayanan [2019], to be useful for understanding terms of measure type, but we should not be too loose with it, because it is possible to define integration operators that do not correspond to quasi-Borel measures.)

Measure terms as samplers. We can understand terms of measure type, built using **sample**, **score**, **return**, and $\mathcal{M}$.**do**, not only as denoting abstract mathematical measures, but alternatively as denoting samplers. Intuitively, for example, we might wish to think of **sample** statements as generating random numbers on the unit interval, **score** statements as accumulating an importance weight as a program executes, **return** as deterministically returning a certain value, and $\mathcal{M}$.**do**$\{x \leftarrow \mu; t\}$ as sampling from μ, assigning to the variable x the sampled value, and continuing to execute the instructions in t. Ścibior et al. [2017] introduce the notion of an *inference representation* to formalize this idea. An inference representation represents a strategy for interpreting code of measure type: for example, there is an inference representation for importance samplers, which produce weighted samples as described above, and another one for population samplers, which implement particle filters by maintaining lists of weighted samples. We refer the reader to their work for a formal presentation; for the purposes of our paper, it suffices to note that any term of type $\mathcal{M}\,\tau$ can be understood as implementing a sampler, and that terms that denote the same measure may nevertheless correspond to samplers with very different performance characteristics (e.g., convergence rates). Indeed, in programmable inference, our goal is often to construct samplers that target the same measure as would a generic inference algorithm, but which converge more quickly to the target.

5 PROBABILISTIC PROGRAMMING WITH TRACE TYPES

We now extend the language of Ścibior et al. [2017] with constructs for probabilistic programming with trace types. Rather than constructing samplers (of type $\mathcal{M}\,\tau$) directly, users of our extended language can choose to first construct generative models and auxiliary probabilistic programs (proposals, kernels, or variational families), of *traced probabilistic program* type ($\mathcal{P}\;\tau\;\alpha$), before invoking sound-by-construction programmable inference constructs (Section 6) to obtain

samplers. We design our language for these programs to capture key features of systems like Gen [Cusumano-Towner et al. 2019] and Pyro [Bingham et al. 2019], although our type system necessarily introduces some additional restrictions; we evaluate the expressiveness of the proposed modeling language in Section 5.3.

5.1 Stock measures, densities, and primitive distributions

The language of samplers developed in Section 4 had a single **sample** statement, for sampling uniform random numbers on the unit interval. In our language for traced probabilistic programs, by contrast, users will instead sample from a set of primitive distributions that come equipped with densities and guarantees about their supports. In this section, we add these primitive distributions to the language; to do so, we first need to define the concepts of *density* and *absolute continuity*.

Density and absolute continuity. Given two measures $\mu, \nu : \mathcal{M}\,\tau$, we say that $\rho : \tau \to \mathbf{R}_{\geq 0}$ is a *density* (sometimes called a *Radon-Nikodym derivative*) of μ with respect to ν if $[\![\mu]\!] = [\![\mathcal{M}.\mathbf{do}\,\{x \leftarrow \nu;\ \mathbf{score}\,\rho(x);\ \mathbf{return}\,x\}]\!]$. When such a density exists, we say that μ is *absolutely continuous* with respect to ν, and write $\mu \ll \nu$. (When μ and ν are probability measures, one intuitive understanding of absolute continuity is that if μ assigns some probability to an event, then so must ν: otherwise, no matter how big ρ is, its integral over that event with respect to ν will be 0. For example, taking **lebesgue** to be the Lebesgue measure on $\mathbb{R}$—as an integrator, $\lambda f.\int_{\mathbb{R}} f(x)dx$—note that **return** 7 is *not* absolutely continuous with respect to **lebesgue**, as **lebesgue** assigns measure 0 to the set $\{7\}$.)

Stock measures. We'd like to equip each primitive distribution in our language with a density—but a density with respect to what measure? To resolve this question, we assign to each ground type τ in our language a *stock measure* $\mathcal{B}[\![\tau]\!]$. For ground types τ with countably many elements (**1**, **N**, and **Fin** n), we set $\mathcal{B}[\![\tau]\!]$ to $\mathbf{counting}_\tau$, the counting measure on τ. As an integrator, the counting measure maps functions $f : \tau \to \overline{\mathbf{R}}_{\geq 0}$ to their sums over all the elements of $[\![\tau]\!]$: $\mathbf{counting}_\tau = \lambda f.\sum_{x \in [\![\tau]\!]} f(x)$. For ground types τ that are subsets of **R** (**U**, $\mathbf{R}_+$, etc.), we set $\mathcal{B}[\![\tau]\!]$ to $\mathbf{lebesgue}_\tau$, the Lebesgue measure on $[\![\tau]\!]$. As an integrator, $\mathbf{lebesgue}_\tau = \lambda f.\int_{[\![\tau]\!]} f(x)dx$, that is, it takes a classical Lebesgue integral. The partial function $\mathcal{B}[\![\cdot]\!]$ from types to their stock measures can be extended to cover product types and sum types (and thus lists, vectors, and records), using the standard constructions for product measures and sum measures. It does not cover function or measure types.

Density-carrying measures. For each type τ equipped with a stock measure $\mathcal{B}[\![\tau]\!]$, we now add types of *density-carrying measures* over τ: the type $\mathcal{U}\,\tau$ contains measures that have strictly positive densities with respect to $\mathcal{B}[\![\tau]\!]$, and $\mathcal{D}\,\tau$ represents the subset of those that are probability measures. See Figure 7 for details. For a density-carrying measure d, the accessor $d.sampler$ extracts the underlying sampler, and $d.density$ the strictly positive density.

The strictly positive density condition ensures that not only is every density-carrying measure d over τ absolutely continuous with respect to the stock measure $\mathcal{B}[\![\tau]\!]$ (by definition of absolute continuity), but we also have the reverse ($\mathcal{B}[\![\tau]\!] \ll d.sampler$). Why? If ρ is a strictly positive density of μ with respect to ν, then its pointwise inverse $\lambda t.\frac{1}{\rho(t)}$ is a strictly positive density of ν with respect to μ. Since absolute continuity is transitive, *all* density-carrying measures on a common type τ are mutually absolutely continuous: for any such d_1 and d_2, $d_1.sampler \ll \mathcal{B}[\![\tau]\!] \ll d_2.sampler$.

A consequence of the strictly positive density requirement is that the type τ must precisely characterize the support of the distribution. Primitive distributions must therefore be assigned carefully chosen types. For example, categorical distributions have types of the form $\mathcal{D}\,(\mathbf{Fin}\,n)$, not $\mathcal{D}\,\mathbf{N}$, and we distinguish between real-valued distributions supported on the entire real line

Type	Gloss	Denotation
$\mathcal{U}\,\tau$	density-carrying measures over τ	The space of pairs $(\mathit{sampler}, \mathit{density}) \in [\![\mathcal{M}\,\tau \times (\tau \to \mathbf{R}_+)]\!]$ such that *density* is a density of *sampler* with respect to $\mathcal{B}[\![\tau]\!]$.
$\mathcal{D}\,\tau$	density-carrying probability distributions over τ	The space of pairs $(\mathit{sampler}, \mathit{density}) \in [\![\mathcal{U}\,\tau]\!]$ such that *sampler* is a probability measure.
$\mathcal{P}\,\tau\,\alpha$	traced probabilistic programs with trace type $\tau = \prod_{i\in I}\tau_i$ and return type α	The space of tuples $(\mathit{traced}, \mathit{returnValue}, \mathit{observe}_\emptyset, \ldots, \mathit{observe}_I)$, where $\mathit{traced} \in [\![\mathcal{D}\,\tau]\!]$ is a density-carrying probability distribution over traces, $\mathit{returnValue} \in [\![\tau \to \alpha]\!]$ is a deterministic function mapping traces to program return values, and for each set of labels $J \subseteq I$, $\mathit{observe}_J \in [\![\prod_{j\in J}\tau_j \to \mathcal{U}\,(\prod_{i\in I\setminus J}\tau_i)]\!]$ is such that $\lambda t.(\mathit{observe}_J\ t).\mathit{sampler}$ is a trace disintegration of $\mathit{traced.sampler}$ with respect to $\prod_{j\in J}\tau_j$.

Fig. 7. New types and denotations for probabilistic programs

(**normal** $\mu\,\sigma$, for example), and those supported only on the positive reals (**gamma** $a\,b$) or on an interval (**uniform**).

5.2 Traced probabilistic programs

We now embed into our core calculus a language for traced probabilistic programs, the Structured Trace Probabilistic Language (STPL). It includes constructs for named sampling from primitive distributions, sequencing via monadic composition, and several more advanced control flow features, enabling restricted forms of stochastic branching and looping. As these programs are built up, we construct types that precisely characterize the space of their execution traces, and density functions for scoring those traces, which serve both as evidence of absolute continuity with respect to other probabilistic programs of the same type, and as practical tools that will be useful for implementing inference algorithms. The traces themselves are records, mapping the names of sample statements in the program to their realized values in a particular execution.

Formally, an STPL program is a term of type $\mathcal{P}\,\tau\,\alpha$, a probabilistic program with trace type τ and return type α. As detailed in Figure 7, such a program p comes equipped with a density-carrying probability distribution over traces, $p.\mathit{traced}$, a valuation function $p.\mathit{returnValue}$ taking traces to return values, and a collection of *trace disintegrations*, $p.\mathit{observe}_J$, for conditioning a subset J of the program's traced random variables. These disintegrations are in lieu of the traditional **score** statement; by excluding **score** from the language, we ensure that traced probabilistic programs (which may represent proposal distributions or simulators for amortized inference) can be sampled from exactly. Disintegration is a measure-theoretic notion that formalizes the intuitive notion of conditioning on observations [Chang and Pollard 1997; Narayanan 2019]. Supposing $\mu : \mathcal{M}\,(\tau + \sigma)$ is a measure over a record type with components τ and σ, we define a *trace disintegration* of μ with respect to τ to be a function $\mu_\tau : \tau \to \mathcal{M}\,\sigma$ satisfying $[\![\mu]\!] = [\![\mathcal{M}.\mathbf{do}\,\{t \leftarrow \mathcal{B}[\![\tau]\!]; s \leftarrow \mu_\tau\ t; \mathbf{return}\ t + s\}]\!]$. Intuitively, $\mu_\tau\ t$ represents an unnormalized posterior over any *latent* variables not included in the query trace t of observations.

We now describe STPL's programming constructs; complete typing rules are given in Figure 8.

5.2.1 Deterministic computation. We first define $\mathcal{P}.\mathbf{return}\,x$ (Figure 9, top left) to combine the Dirac distribution on the empty trace, **return** $\{\}$, with a constant return value function. The traces of STPL programs record only the random choices they make, and $\mathcal{P}.\mathbf{return}$ is completely deterministic.

$$\frac{\Gamma \vdash d : \mathcal{D}\,\tau \qquad l \in \mathcal{L}}{\Gamma \vdash \mathbf{sample}_l\ d : \mathcal{P}\,\{l : \tau\}\,\tau} \qquad \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash \mathcal{P}.\mathbf{return}\ t : \mathcal{P}\,\{\}\,\alpha} \qquad \frac{}{\Gamma \vdash \mathcal{B}[\![\tau]\!] : \mathcal{M}\,\tau} \qquad \frac{\Gamma \vdash p : \mathcal{P}\,\tau\,\alpha}{\Gamma \vdash p.traced : \mathcal{D}\,\tau}$$

$$\frac{\Gamma \vdash \mu : \mathcal{P}\,\tau\,\alpha \qquad \Gamma, x : \alpha \vdash \nu : \mathcal{P}\,\sigma\,\beta \qquad \text{labels}(\tau) \cap \text{labels}(\sigma) = \emptyset}{\Gamma \vdash \mathcal{P}.\mathbf{do}\,\{x \leftarrow \mu; \nu\} : \mathcal{P}\,(\tau +\!\!+ \sigma)\,\beta} \qquad \frac{\Gamma \vdash p : \mathcal{P}\,\tau\,\alpha}{\Gamma \vdash p.returnValue : \tau \to \alpha}$$

$$\frac{\Gamma \vdash p : \mathcal{P}\,\tau\,\alpha \qquad \Gamma \vdash q : \mathcal{P}\,\sigma\,\alpha \qquad \Gamma \vdash u : \mathbf{U} \qquad l \in \mathcal{L}}{\Gamma \vdash \mathbf{withProbability}_l\ u\ p\ \mathbf{else}\ q : \mathcal{P}\,\{l : \tau + \sigma\}\,\alpha} \qquad \frac{\Gamma \vdash p : \mathcal{P}\,(\tau +\!\!+ \sigma)\,\alpha \qquad \Gamma \vdash t : \tau}{\Gamma \vdash p.observe\,t : \mathcal{U}\,\sigma}$$

$$\frac{\Gamma \vdash d : \mathcal{D}\,\mathbf{N} \qquad \Gamma, i : \mathbf{N} \vdash p : \mathcal{P}\,\tau\,\alpha \qquad l \in \mathcal{L}}{\Gamma \vdash \mathbf{for}_l\ i\ \mathbf{inRandomRange}\ d\ p : \mathcal{P}\,\{l : \mathbf{List}\,\tau\}\,(\mathbf{List}\,\alpha)} \qquad \frac{\Gamma \vdash d : \mathcal{D}\,\tau \text{ or } \mathcal{U}\,\tau}{\Gamma \vdash d.density : \tau \to \mathbf{R}_+}$$

$$\frac{\Gamma \vdash t : \mathbf{Vec}\,n\,\alpha \qquad \Gamma, x : \alpha \vdash p : \mathcal{P}\,\tau\,\beta \qquad n \in \mathbb{N} \qquad l \in \mathcal{L}}{\Gamma \vdash \mathbf{foreach}_l\ x\ \mathbf{in}\ t\ p : \mathcal{P}\,\{l : \mathbf{Vec}\,n\,\tau\}\,(\mathbf{Vec}\,n\,\beta)} \qquad \frac{\Gamma \vdash d : \mathcal{D}\,\tau \text{ or } \mathcal{U}\,\tau}{\Gamma \vdash d.sampler : \mathcal{M}\,\tau}$$

$$\frac{\Gamma \vdash s_0 : \alpha \qquad \Gamma \vdash p_{max} : \mathbf{U} \qquad \Gamma, s : \alpha \vdash p : \mathbf{U} \qquad \Gamma, s : \alpha \vdash q : \mathcal{P}\,\tau\,\alpha \qquad l \in \mathcal{L}}{\Gamma \vdash \mathbf{while}_l\,(s\ :=\ s_0;\ min(p, p_{max}))\ q : \mathcal{P}\,\{l : \mathbf{List}\,\tau\}\,\alpha}$$

Fig. 8. Type system for probabilistic programs and distributions.

5.2.2 Sampling at an address. We next add a construct, $\mathbf{sample}_l\ d$, that turns any primitive distribution $d : \mathcal{D}\,\alpha$ into a probabilistic program of type $\mathcal{P}\,\{l : \alpha\}\,\alpha$ that samples from the distribution, traces the result at a particular label $l \in \mathcal{L}$, and returns the sampled value (Figure 9, top right).

5.2.3 Sequencing statements. We next give our family of types $\mathcal{P}$ constructs for monadic composition. $\mathcal{P}$ does not form a monad in the classical sense, but does form a *graded* monad [Fujii et al. 2016; Katsumata 2014], and we can still take advantage of **do**-notation (Figure 9, bottom).

Like the traditional monadic bind, our version, $\mathcal{P}.\mathbf{do}\,\{x \leftarrow \mu; \nu\}$, sequences two probabilistic programs, binding the result of sampling from μ to the variable x, then running the remainder of the program, ν. The resulting probabilistic program has an expanded trace type: its trace is the concatenation of the traces of μ and ν. In order for this to make sense, we need to ensure that the label sets of τ and σ are disjoint, a condition that appears in the $\mathcal{P}.\mathbf{do}$ typing rule in Figure 8.

Although this typing rule is standard, it has an important consequence that is crucial to our approach. In particular, the premise $\Gamma, x : \alpha \vdash \nu : \mathcal{P}\,\sigma\,\beta$ ensures that *no matter the value x of type α* produced by the first part of a probabilistic program, the rest of the program always has the same trace type σ. This implies that the rest of the program, when viewed as a distribution over traces, has the same support no matter what happened in the first part of the program (though of course, it may assign very low probability to some traces). Informally, we might say that in a probabilistic program built up in this way, *no matter what happened in the past, anything can still happen.* This is why our control flow structures, which we introduce next, must all be stochastic, assigning some probability (however small) to every possible control flow path. This is a restriction on the expressivity of our language that allows us to reason more carefully about stochastic control flow; we evaluate the severity of this restriction and others in Section 5.3.

5.2.4 Branching. Programmers can branch on Boolean values, randomly sampled or otherwise, using **if** from the core calculus. But because probabilistic programs that make different named

$$\mathcal{P}.\mathbf{return}\ a$$

$$\begin{aligned}
traced.sampler &= \mathbf{return}\ \{\} \\
traced.density(t) &= 1 \\
returnValue(t) &= a \\
observe_{\emptyset}(t) &= traced
\end{aligned}$$

$$\mathbf{sample}_l\ d$$

$$\begin{aligned}
traced.sampler &= \mathcal{M}.\mathbf{do}\ \{\ x \leftarrow d.sampler; \mathbf{return}\ \{l = x\}\} \\
traced.density(t) &= d.density(t.l) \\
returnValue(t) &= t.l \\
observe_{\emptyset}(t) &= traced \\
observe_{\{l\}}(t).sampler &= \mathcal{M}.\mathbf{do}\ \{\mathbf{score}\ (d.density(t.l));\ \mathbf{return}\ \{\ \}\} \\
observe_{\{l\}}(t).density(u) &= d.density(t.l)
\end{aligned}$$

$$\begin{aligned}
&traced.sampler = \mathcal{M}.\mathbf{do}\ \{ \\
&\quad t_\tau \leftarrow \mu.traced.sampler \\
&\quad \mathbf{let}\ x = \mu.returnValue(t_\tau) \\
&\quad t_\sigma \leftarrow (\nu\ x).traced.sampler \\
&\quad \mathbf{return}\ t_\tau +\!\!+ t_\sigma \\
&\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{P}.\mathbf{do}\ \{x \leftarrow \mu; \nu(x)\} \\
&\qquad observe_K(t).sampler = \mathcal{M}.\mathbf{do}\ \{ \\
&\qquad\quad \mathbf{let}\ (t_\tau, t_\sigma) = (\mathbf{restrict}_{K\cap\tau}(t), \mathbf{restrict}_{K\cap\sigma}(t)) \\
&\qquad\quad s_\tau \leftarrow (\mu.observe_{K\cap\tau}(t_\tau)).sampler \\
&\qquad\quad \mathbf{let}\ x = \mu.returnValue(s_\tau +\!\!+ t_\tau) \\
&\qquad\quad s_\sigma \leftarrow ((\nu\ x).observe_{K\cap\sigma}(t_\sigma)).sampler \\
&\qquad\quad \mathbf{return}\ s_\tau +\!\!+ s_\sigma \\
&\qquad \}
\end{aligned}$$

$$\begin{aligned}
&returnValue(t) = (\nu\ (\mu.returnValue(\mathbf{restrict}_\tau\ t))).returnValue(\mathbf{restrict}_\sigma\ t) \\
&observe_K(t).density(s) = \\
&\quad \mu.observe_{K\cap\tau}(\mathbf{restrict}_{K\cap\tau}\ t).density(\mathbf{restrict}_{\tau\setminus K}\ s) * \\
&\quad (\nu\ (\mu.returnValue(\mathbf{restrict}_\tau(t +\!\!+ s)))).observe_{K\cap\sigma}\ (\mathbf{restrict}_{K\cap\sigma}(t)).density(\mathbf{restrict}_{\sigma\setminus K}(s)) \\
&traced.density(t) = \mu.traced.density(\mathbf{restrict}_\tau\ t) * \\
&\qquad\qquad (\nu\ \mu.returnValue(\mathbf{restrict}_\tau\ t)).traced.density(\mathbf{restrict}_\sigma\ (t))
\end{aligned}$$

Fig. 9. Implementations of **return**, $\mathbf{sample}_l$, and $\mathcal{P}.\mathbf{do}$

choices have different types in our language, they cannot appear on two sides of an **if**: the default branching constructs can only be used to change the values flowing through a program and the distributions from which random choices are sampled, not the structure of a program's trace.

We recover some of this lost flexibility with a new branching construct, $\mathbf{withProbability}_l$, which tosses a biased coin to decide whether to execute one piece of generative code or another. The resulting trace contains a sum type (Figure 8). Recall that the sum type $\tau + \sigma$ is different from $\tau +\!\!+ \sigma$, which is the concatenation of two record types. The addresses of τ and σ here can overlap. The enforcement that the choice of which branch to go down be random (our **U** type does not include 0 or 1) ensures that our density is strictly positive, for any trace of type $\{l : \tau + \sigma\}$: no region of the trace space with non-zero base measure is assigned zero probability.

5.2.5 Iteration. We introduce three distinct constructs for iteration. The first, $\mathbf{foreach}_l$, iterates over a vector with statically known length. The trace of the resulting program includes at the label l a vector of the same length, containing one subtrace (of the loop body) per iteration. Next, we introduce $\mathbf{for}_l$ loops that choose a number of iterations randomly from a distribution on $\mathbb{N}$. In this case, the traces contain lists of subtraces, rather than vectors. We know that all possible list lengths have positive probability, because our definition of the type $\mathcal{D}\,\mathbf{N}$ requires a strictly positive density with respect to $\mathcal{B}[\![\mathbf{N}]\!] = \mathbf{counting}_{\mathbf{N}}$. Finally, we add a $\mathbf{while}_l$ loop. Speaking operationally, the loop begins by initializing a state variable s to a, then computing p as a function of the state, which

yields a *probability of continuing*. With probability $min(p, p_{max})$, we enter the loop body q, adding its trace to our list of subtraces, and updating the state variable s with its return value. We then evaluate p again, and flip another coin. When we hit our first tails, the loop ends. The requirement to specify a maximum probability-of-continuing ensures that the loop halts with probability 1, which is necessary for ensuring that our density calculation is correct.

With these constructs defined, we can now state an important theorem:

Theorem 1. $\mathcal{P}.\textbf{\textit{return}}$, $\textbf{\textit{sample}}_l$, $\mathcal{P}.\textbf{\textit{do}}\{x \leftarrow \mu; \nu\}$, $\textbf{\textit{withProbability}}_l$, $\textbf{\textit{foreach}}_l$, $\textbf{\textit{for}}_l$, *and* $\textbf{\textit{while}}_l$ *construct bona-fide traced probabilistic programs, satisfying the conditions of the definition in Figure 7.*

The proof is by induction, taking as base cases **return** and **sample** (Figure 9). The densities for monadic composition (also shown in Figure 9) and the control flow constructs (given in the supplement) are built from the densities of the component programs on which they act, using only operations (like multiplication) that preserve strict positivity. We verify the correctness of the densities and disintegrations via equational reasoning, using the commutativity of the monad $\mathcal{M}$ and identities like $[\![\mathcal{M}.\textbf{do}\{\textbf{score}\, x; \textbf{score}\, y\}]\!] = [\![\textbf{score}\,(x * y)]\!]$. See supplement for details.

Theorem 1 ensures that probabilistic programs written with these primitives come with correct-by-construction density calculation procedures, and that their trace distributions are absolutely continuous with respect to a common base measure. It also implies their trace disintegrations are really disintegrations, so that users' conditioning queries are interpreted in a sensible way.

We note briefly that the type system described in this section uses record types with type-level record concatenation and disjointness checking, not common features in today's popular functional programming languages. We implemented a prototype of our type system in Haskell using the row-types package,[1] and found that on our examples, the types of traced probabilistic programs could be automatically inferred by GHC. Error messages for ill-typed code were somewhat informative, but the prototype is still a bit unpleasant to use, partly because of the awkwardness of dealing with dependent types in Haskell. We do see the prototype as evidence, however, that a custom type inference engine for probabilistic code may be relatively straight-forward to implement.

5.3 Expressiveness of the typed language

STPL supports higher-order functions and restricted forms of branching and loops, but is not Turing-complete. (For one thing, all programs halt with probability 1.) Furthermore, STPL disallows **score** statements, which means all conditioning must be done via the built-in disintegration functions $p.observe$. We now discuss and evaluate the severity of these restrictions.

- STPL does not support **score** statements in models. But for positive w, we can mimic a **score** w statement by sampling from an exponential distribution with rate parameter w, then, at inference time, calling *observe* with an observation trace that constrains the exponential sample to 0. Using this trick, we can mimic a **score** statement in any program meant to encode a *model*, but not a proposal, transition kernel, or variational family. This restriction is intentional, as making use of transition kernels and variational families with **score** statements can require potentially intractable integration.
- All traced samples must be from distributions with statically-known support. But users can mimic samples from, e.g., **uniform** $a\, b$ by sampling from a uniform distribution on $(0, 1)$ and scaling and translating the result appropriately. So the only real restriction here is that the user cannot *condition* a model on the result of a sample whose support is dynamic.
- Our control-flow constructs cannot encode models in which arbitrary deterministic functions of random choices may change the program's support. For example, we cannot sample n,

[1] http://hackage.haskell.org/package/row-types

then branch on whether n is prime, or loop for a number of iterations equal to the n^{th} prime number (unless the branch or loop bodies are deterministic). Using **withProbability** or a **while** loop, however, we can get arbitrarily close to this, by implementing a stochastic version of the model that behaves according to deterministic rules with arbitrarily high probability.
- As presented, our looping constructs force proposals, kernels, and variational families to sample choices in loop bodies in the same order-of-iterations that the model does. We could add *trace-permuting loops*, which apply a dynamically computed permutation to the list of subtraces produced by a loop before including the list in a trace, to address this issue.

How severe are these restrictions? We have evaluated online listings of example models for two popular probabilistic programming languages (Turing and Anglican) and found that 75% of example models were expressible in our framework; the inexpressible models included nonparametric models involving Dirichlet Processes, rejection sampling using Dirac deltas within the model, or, in one case, general recursion. We believe this stems from the fact that many real-world uses of modeling and inference do not require particularly sophisticated stochastic control flow; when stochastic control flow is required, branching with some probability or looping for a random number of iterations (sampled from a simple primitive distribution) is often enough to capture the phenomenon being modeled. But of course, there are exceptions. We note that in our proposed architecture, programmers need not do all their programming in STPL; they can mix and match, writing parts of their models as traced probabilistic programs, and parts that require more flexibility directly as samplers. For the pieces written in STPL, programmable inference features can be applied to make inference more efficient or accurate. We turn to these features next.

6 SOUND PROGRAMMABLE INFERENCE

In this section, we extend our language with provably sound implementations of constructs for programmable inference (see Figure 2, bottom, for an overview of the new syntax).

6.1 Importance sampling

The simplest form of programmable inference we develop is *custom-proposal importance sampling* (Figure 10). In order to sample approximately from an intractable posterior distribution, importance sampling draws samples instead from a tractable *proposal distribution*, then weights these samples to correct for the discrepancy, using a *density ratio*: if p is the (potentially unnormalized) density of the posterior distribution, and q is the density of the proposal, then importance sampling draws a

Importance sampling with a custom proposal

$$\frac{\Gamma \vdash p : \mathcal{P}\,(\tau +\!\!+ \sigma)\,\alpha \qquad \Gamma \vdash t : \tau \qquad \Gamma \vdash q : \mathcal{P}\,\sigma\,\mathbf{1}}{\Gamma \vdash \textbf{importance}\ p\ t\ q : \mathcal{M}\,\sigma}$$

```
1  importance p t q = M.do {
2    t_q ← q.traced.sampler
3    score ((p.observe t).density t_q / q.traced.density t_q)
4    return t_q
5  }
```

For all proposals q with trace type σ, and all constraint traces t, we have:

$$[\![\textbf{importance}\ p\ t\ q]\!] = [\![(p.\mathit{observe}\ t).\mathit{sampler}]\!]$$

Fig. 10. Typing rule, implementation, and soundness theorem for custom-proposal importance sampling

sample $x \sim q$ and computes the weight $\frac{p(x)}{q(x)}$. Importance sampling can fail if q is chosen poorly: it must, for example, have a chance of sampling x's from the entire support of p (i.e., we require absolutely continuity, $p \ll q$). Importance sampling can also fail if the densities of p and q are computed with respect to different base measures: then the ratio does not actually represent the Radon-Nikodym derivative of p with respect to q, as it must for importance sampling to be sound.

Our type system ensures that the unnormalized posterior $p.observe_\tau\ t$ is absolutely continuous with respect to q, because both have trace type σ. To prove the soundness theorem, we begin by unfolding the definition in Figure 10, then reason equationally:

$$
\begin{aligned}
&[\![\mathcal{M}.\mathbf{do}\,\{t_q \leftarrow q.traced.sampler; \mathbf{score}\left(\frac{(p.observe\,t).density\,t_q}{q.traced.density\,t_q}\right); \mathbf{return}\,t_q\}]\!] \\
&= [\![\mathcal{M}.\mathbf{do}\,\{t_q \leftarrow \mathcal{B}[\![\sigma]\!]; \mathbf{score}\,(q.traced.density\,t_q); \mathbf{score}\left(\frac{(p.observe\,t).density\,t_q}{q.traced.density\,t_q}\right); \mathbf{return}\,t_q\}]\!] \\
&= [\![\mathcal{M}.\mathbf{do}\,\{t_q \leftarrow \mathcal{B}[\![\sigma]\!]; \mathbf{score}\,((p.observe\,t).density\,t_q); \mathbf{return}\,t_q\}]\!] \\
&= [\![\mathcal{M}.\mathbf{do}\,\{t_q \leftarrow (p.observe\,t).sampler; \mathbf{return}\,t_q\}]\!] = [\![(p.observe\,t).sampler]\!]
\end{aligned}
$$

The first and third equations make use of the definition of *density*, and the fact that our density-carrying measures (including $q.traced$ and $p.observe\,t$) know their own densities with respect to the stock measure $\mathcal{B}[\![\sigma]\!]$. The second equation follows from the fact that $[\![\mathcal{M}.\mathbf{do}\,\{\mathbf{score}\,x;\ \mathbf{score}\,y\}]\!] = [\![\mathbf{score}(x * y)]\!]$. In the last equation, we apply the monad laws.

This kind of reasoning, used extensively by Narayanan [2019] to reason about meaning-preserving program transformations for probabilistic code, is the workhorse of all our proofs. Because *bind* in the monad of measures represents integration, the proof above is really just another way of representing measure-theoretic arguments: the first equation, for example, rewrites an integral with respect to the probability measure $q.traced.sampler$ (i.e., an expectation) as an integral with respect to a stock measure, where the integrand has been multiplied by the density.

6.2 Sequential Monte Carlo

We now turn to a slightly more sophisticated method: particle filtering, a form of sequential Monte Carlo. Particle filtering can be used in state-space models, which start by initializing some state, then iterate a stochastic transition, producing new latent variables and observed variables at each iteration. The combinator **unroll**, shown below, takes in a state-initializing probabilistic program $init : \mathcal{P}\,\tau\,\alpha$, as well as a transition program $next : \alpha \to \mathcal{P}\,(\sigma +\!\!+ \gamma)\,\alpha$, and a list $steps : \mathbf{List}\,\gamma$ of observations from each time step. It produces a sampler $p : \mathcal{M}\,(\tau \times \mathbf{List}\,\sigma)$ targeting the posterior measure over the unobserved random variables: the state-initializer's trace, and the σ-shaped traces of latent variables from each time step. Its code is shown below:

```
1  unroll init next steps = M.do {
2    t_init ← init.traced.sampler
3    let processStep = λ soFar t. M.do {
4      (s, results) ← soFar
5      t_next ← ((next s).observe t).sampler
6      return ((next s).returnValue t_next, Cons t_next results)
7    }
8    (s, l) ← fold steps (return(init.returnValue t_init, Nil)) processStep
9    return(t_init, l)
10 }
```

Particle filtering with custom proposals

$$\frac{\Gamma \vdash q_{init} : \mathcal{P}\,\tau\,\mathbf{1} \qquad \Gamma \vdash init : \mathcal{P}\,\tau\,\alpha \qquad \Gamma \vdash next : \alpha \to \mathcal{P}\,(\sigma +\!\!+ \gamma)\,\alpha \qquad \Gamma \vdash steps : \mathbf{List}\,(\sigma \times (\alpha \to \mathcal{P}\,\gamma\,\mathbf{1}))}{\Gamma \vdash \mathbf{particleFilter}\ init\ q_{init}\ next\ steps : \mathcal{M}\,(\tau \times \mathbf{List}\,\gamma)}$$

```
1  particleFilter init q_init next steps = M.do {
2    t_init ← importance init { } q_init
3    let processStep = λ soFar (t, q). M.do {
4        (s, results) ← soFar
5        t_next ← importance (next s) t (q s)
6        return ((next s).returnValue t_next, Cons t_next results)
7    }
8    (s, l) ← fold steps (return(init.returnValue t_init, Nil)) processStep
9    return(t_init, l)
10 }
```

When the terms involved are well-typed, we have:

$$[\![\mathbf{particleFilter}\ init\ q_{init}\ next\ steps]\!] = [\![\mathbf{unroll}\ init\ next\ (\mathbf{map}\ fst\ steps)]\!]$$

Fig. 11. Type, implementation, and soundness theorem for custom-proposal particle filtering

The sampler returned by **unroll** implements default inference for this model. Instead, we provide the inference command **particleFilter** (Figure 11), which also accepts custom proposals for the initialization and the latent variables at each time step. (It accepts a different proposal at each time step, which allows the proposal's behavior to depend on the observations.)

The resulting sampler, of type $\mathcal{M}\,(\tau \times \mathbf{List}\,\sigma)$, samples from the proposal distributions instead of the model, then uses **score** to weight the result correctly. The real power of this approach is evident when the resulting term is interpreted using an inference representation (Section 4.3) that implements population sampling (the parallel sampling of many "particles") and suspension-and-resampling at **score** statements, in which whenever the weights of the particles change, low-weight particles are culled and replaced with clones of higher-weight particles. Such an inference representation is developed compositionally by Ścibior et al. [2017]; we can import it unchanged here, enabling the use of custom proposals with particle filters that support these features.

6.3 Markov Chain Monte Carlo

To handle Markov Chain Monte Carlo methods, we first introduce a type, $\mathcal{K}\,\tau\,\sigma$, of *stationary kernels* with trace type $\tau = \sigma +\!\!+ \gamma$ and modification set σ. The type $\mathcal{K}\,\tau\,\sigma$ denotes the space of morphisms $\psi : \mathcal{U}\,\tau \to \tau \to \mathcal{M}\,\tau$ such that for any density-carrying measure $p : \mathcal{U}\,\tau$, we have:

- $[\![p.sampler]\!] = [\![\mathcal{M}.\mathbf{do}\,\{t \leftarrow p.sampler; \psi\,p\,t\}]\!]$ (i.e., if the trace t is distributed according to p, then running it through the transition kernel $\psi\,p$ does not change its distribution), and
- $\psi\,p$ leaves the γ part of its input trace unchanged: $[\![\mathcal{M}.\mathbf{do}\,\{t \leftarrow p.sampler; t' \leftarrow \psi\,p\,t; \mathbf{return}\,(\mathbf{restrict}_\gamma\ t = \mathbf{restrict}_\gamma\ t')\}]\!]$ is almost everywhere equal to **True**.

Stationary kernels of a certain trace type stochastically transform traces of that type in such a way that the distribution from which the original trace was drawn is preserved. Our stationary kernels are stationary (distribution-preserving) for *any* properly typed density-carrying measure.

Metropolis-Hastings step with a custom multivariate proposal

$$\frac{\Gamma \vdash q : (\sigma +\!\!+ \gamma) \to \mathcal{P}\ \sigma\ \mathbf{1}}{\Gamma \vdash \mathbf{mh}\ q : \mathcal{K}\ (\sigma +\!\!+ \gamma)\ \sigma}$$

```
1  mh q = λ p old. M.do {
2    proposed ← ψ_q old
3    r ← sample
4    if r < min(1, ρ_q p (old, proposed)) then
5      return proposed
6    else return old
7  }

8  ψ_q old = M.do {
9    new ← (q old).traced.sampler
10   let rest = restrict_γ old
11   return new ++ rest
12 }
```

$$\rho_q\ p\ (x, y) = \frac{(p.density\ y) * (q\ y).traced.density(\mathbf{restrict}_\sigma\ x)}{(p.density\ x) * (q\ x).traced.density(\mathbf{restrict}_\sigma\ y)}$$

when $\mathbf{restrict}_\gamma\ x = \mathbf{restrict}_\gamma\ y$, and 0 otherwise.

For any unnormalized distribution p on traces of type $(\sigma +\!\!+ \gamma)$, the kernel returned by **mh** q is stationary for $[\![p.sampler]\!]$.

Fig. 12. Typing rule, implementation, and soundness theorem for custom-proposal MH step

Composition of stationary kernels for custom MCMC schedules

$$\frac{\Gamma \vdash k_1 : \mathcal{K}\ \tau\ (\prod_I \sigma_i) \qquad \Gamma \vdash k_2 : \mathcal{K}\ \tau\ (\prod_J \sigma_j)}{\Gamma \vdash \mathbf{seq}_\mathcal{K}\ k_1\ k_2 : \mathcal{K}\ \tau\ \prod_{I \cup J} \sigma_j}$$

$$\frac{\Gamma \vdash p : \mathbf{U} \qquad \Gamma \vdash k_1 : \mathcal{K}\ \tau\ (\prod_I \sigma_i) \qquad \Gamma \vdash k_2 : \mathcal{K}\ \tau\ (\prod_J \sigma_j)}{\Gamma \vdash \mathbf{mix}_\mathcal{K}\ p\ k_1\ k_2 : \mathcal{K}\ \tau\ \prod_{I \cup J} \sigma_j}$$

$$\frac{\Gamma \vdash n : \mathbf{N} \qquad \Gamma \vdash k : \mathcal{K}\ \tau\ \sigma}{\Gamma \vdash \mathbf{repeat}_\mathcal{K}\ n\ k : \mathcal{K}\ \tau\ \sigma} \qquad \frac{\Gamma \vdash b : \tau \to \mathbf{B} \qquad \Gamma \vdash k : \mathcal{K}\ (\tau +\!\!+ \sigma)\ \sigma}{\Gamma \vdash \mathbf{if}_\mathcal{K}\ b\ k : \mathcal{K}\ \tau\ \sigma}$$

$$\begin{aligned}
\mathbf{seq}_\mathcal{K}\ k_1\ k_2 &= \lambda\, \mu\, t.\ \mathcal{M}.\mathbf{do}\ \{t' \leftarrow\ k_1\ \mu\ t;\ k_2\ \mu\ t'\} \\
\mathbf{mix}_\mathcal{K}\ p\ k_1\ k_2 &= \lambda\, \mu\, t.\ \mathcal{M}.\mathbf{do}\ \{u \leftarrow \mathbf{sample};\ (\mathbf{if}\ u < p\ \mathbf{then}\ k_1\ \mathbf{else}\ k_2)\ \mu\ t\} \\
\mathbf{repeat}_\mathcal{K}\ n\ k &= \lambda\, \mu\, t.\ \mathbf{fold}_\mathbf{N}\ n\ (\mathbf{return}\ t)\ (\lambda\, k_i.\ \mathcal{M}.\mathbf{do}\ \{t' \leftarrow k_i; k\ \mu\ t'\}) \\
\mathbf{if}_\mathcal{K}\ b\ k &= \lambda\, \mu\, t.\ \mathbf{if}\ (b\ t)\ \mathbf{then}\ k\ \mu\ t\ \mathbf{else\ return}\ t \\
\mathbf{applyKernel}\ \mu\ k\ n &= \mathcal{M}.\mathbf{do}\ \{t \leftarrow \mu;\ \mathbf{repeat}_\mathcal{K}\ n\ k\ \mu\ t\}
\end{aligned}$$

Kernels of type $\mathcal{K}\ \tau\ \sigma$ produced by these kernel combinators are stationary for any unnormalized distribution μ on traces of type τ.

Fig. 13. Typing rules, implementations, and soundness theorem for kernel combinators

Furthermore, they promise (with probability 1) to change at most a subset (determined by σ) of the addresses in the trace; we take advantage of this to introduce an $\mathbf{if}_{\mathcal{K}}$ combinator (Figure 13), which modifies a kernel k to execute only when a certain condition is satisfied on the γ part of the input trace. If k could change γ, such a modified kernel would not necessarily inherit the base kernel k's stationarity property.

Our support for programmable MCMC comes in two flavors: the **mh** operator (Figure 12) turns generative code for stochastically proposing changes to an input trace into a stationary kernel, by adding an Metropolis-Hastings accept-reject step with the properly computed acceptance probability. The kernel combinators (Figure 13) allow users to mix and match stationary kernels, crafting custom update schedules for an MCMC algorithm tailored to their specific problem.

To show that **mh** kernels are stationary, we apply a version of the Metropolis-Hastings theorem formulated in greater generality by Green [1995], and then ported to the quasi-Borel setting by Ścibior et al. [2017].

Theorem 2. *Let $p : \mathcal{U}\,(\sigma + \gamma)$ be a density-carrying measure on traces, and $q : (\sigma + \gamma) \to \mathcal{P}\,\sigma\,\mathbf{1}$ be a Metropolis-Hastings proposal. Then in $\boldsymbol{mh}\,q\,p$, shown in Figure 12, the conditions of the Metropolis-Hastings-Green Theorem are satisfied:*

(1) for all $t : \sigma + \gamma$, $\psi_q\,t$ is a probability measure,
(2) for all x and y such that $(\rho_q\,p)(x,y) \neq 0$, $1 = (\rho_q\,p)(x,y) * (\rho_q\,p)(y,x)$,
(3) $\rho_q\,p$ is a density of $\mu_{swapped} = \mathcal{M}.\boldsymbol{do}\{(t,t') \leftarrow \mu; \boldsymbol{return}\,(t',t)\}$ with respect to μ, where $\mu = \mathcal{M}.\boldsymbol{do}\,\{t \leftarrow p.sampler; t' \leftarrow \psi_q\,p\,t; \boldsymbol{score}\,\mathbf{1}_{\rho_q\,p\,(t,t')\neq 0}; \boldsymbol{return}(t,t')\}$, and
(4) $(\rho_q\,p)(x,y) = 0$ if and only if $(\rho_q\,p)(y,x) = 0$.

Thus, $[\![\boldsymbol{mh}\,q\,p]\!]$ is stationary for $[\![p.sampler]\!]$.

Condition (1) holds because $(q\,t).traced.sampler$ is guaranteed to be a probability measure for all t; ensuring that user-written probabilistic programs like q represent normalized probability measures is one of the reasons that we decided not to include **score** in our language for generative code, instead adding conditioning via disintegration. Conditions (2) and (4) follow by algebraic manipulation. The most involved part of the proof is in showing condition (3). We set as a base measure $\nu = \mathcal{M}.\mathbf{do}\{t \leftarrow \mathcal{B}[\![\tau]\!]; t'_\sigma \leftarrow \mathcal{B}[\![\sigma]\!]; t'_\gamma \leftarrow \mathbf{return}\,(\mathbf{restrict}^\tau_\gamma\,t); \mathbf{return}\,(t, t'_\sigma + t'_\gamma)\}$. We then compute densities of μ and $\mu_{swapped}$ with respect to this base measure, using equational reasoning, and show that their ratio is equal to $\rho_q\,p$ from Figure 12.

6.4 Variational inference

Finally, we introduce typing rules for *variational inference* operations (Figure 14). Unlike in previous sections, we do not give precise meanings and implementations to the variational inference operators we introduce: in this paper, we do not address the semantics of automatic differentiation or differentiability. However, our typing rules ensure that certain preconditions for the sound use of variational inference are satisfied. Anecdotally, we have found that the sorts of bugs these typing rules prevent users from making can be tricky to catch without static checking, especially in bigger programs. For example, in Pyro, if two probabilistic programs sample at misaligned trace addresses, it is possible to train for thousands of iterations with no error indicating anything has gone wrong–despite the fact that the "optimization" the system performs is meaningless.

We introduce two variational inference commands. To use **svi**, as in importance sampling, the programmer passes in a model (of type $\mathcal{P}\,(\tau + \sigma)\,\alpha$) and query trace (of type τ) to specify a target posterior distribution over traces of type σ (the unknown latent variables). But instead of specifying a single proposal distribution, users specify a *variational family*, $q : \Theta \to \mathcal{P}\,\sigma\,\mathbf{1}$, which is parameterized by values of type Θ. The job of **svi** is to optimize the parameters, starting

Variational and amortized inference with custom variational families

$$\frac{\Gamma \vdash p : \mathcal{P}\ (\tau +\!\!+ \sigma)\ \alpha \qquad \Gamma \vdash q : \Theta \to \mathcal{P}\ \sigma\ \mathbf{1} \qquad \Gamma \vdash \theta_{init} : \Theta}{\Gamma \vdash \mathbf{svi}\ p\ t\ q\ \theta_{init} : \mathcal{M}\ \Theta}$$

$$\frac{\Gamma \vdash p : \mathcal{P}\ (\tau +\!\!+ \sigma)\ \alpha \qquad \Gamma \vdash q : \Theta \to \tau \to \mathcal{P}\ \sigma\ \mathbf{1} \qquad \Gamma \vdash \theta_{init} : \Theta}{\Gamma \vdash \mathbf{trainAmortized}\ p\ q\ \theta_{init} : \mathcal{M}\ \Theta}$$

For all $t : \tau$ and $\theta : \Theta$, $p.observe\, t$ and $q\, \theta$ (or, for amortized inference, $q\, \theta\, t$) are mutually absolutely continuous, and compute their densities with respect to the same measure ($\mathcal{B}[\![\sigma]\!]$).

Fig. 14. Typing rules and soundness theorem for variational and amortized inference

with an initial value of θ_{init}, so that $[\![(q\, \theta).traced.sampler]\!]$ is as close as possible, as measured by the *KL divergence*, to the posterior $(p.observe_\tau\, t).sampler$. Importantly, this divergence is infinite if $[\![(q\, \theta).traced.sampler]\!] \not\ll [\![(p.observe_\tau\, t).sampler]\!]$. Our types ensure that absolute continuity holds. (We note, however, that it is still possible to end up with infinite KL for reasons *other* than mismatched supports.)

The second construct, **trainAmortized**, trains the parameters of a program so that it can do inference on *any* query trace of a certain type. To use amortized inference, the user specifies a model p (as before) of type $\mathcal{P}\ (\tau +\!\!+ \sigma)\ \alpha$, and an *inference program* q with two inputs: θ, the parameters being optimized, and τ, a query trace. The inference engine then generates many sample traces from p, breaking each into a pair (τ_i, σ_i). These can then be used as a supervised training set: the parameters θ are optimized so that $q\, \theta$, given τ_i, assigns high probability to σ_i.

Amortized inference requires the joint distribution $p.traced$ over the variables in both τ and σ to be an easy-to-sample probability measure; if we had included **score** in our STPL language, p might not be normalized. Both algorithms discussed in this section (amortized and stochastic variational inference) maximize or minimize objectives involving densities; our constructions ensure that these densities are correctly computed with respect to a common base measure.

7 RELATED WORK

Semantics for validating inference in probabilistic programming. Our denotational approach to validating inference algorithms builds directly on the work of Ścibior et al. [2017], who in turn rely on the semantic foundations laid by Heunen et al. [2018, 2017]. That work addresses the challenge of validating the correctness of Monte Carlo inference algorithms with generic proposals. This paper extends their framework to handle programmable inference algorithms, which use user-written probabilistic programs as proposals, kernels, or variational families. In order to reason about their correctness, we develop (a) semantics for a variety of programmable inference constructs, and (b) a type system that enables us to check statically that the constructs are used soundly.

Our semantics is *intensional*, in the sense that it distinguishes between probabilistic programs that denote the same measures over outputs, by capturing the distribution over a program's possible traces as part of its meaning. Like us, Castellan and Paquet [2019] present an intensional semantics designed to enable the validation of inference algorithms. But their semantics is used to validate an incremental computation scheme for generic Metropolis-Hastings, and as such, tracks different intensional features (e.g., information flow) from ours, which is designed to enable reasoning about

mutual absolute continuity between models and proposals, kernels, and variational families. Our intensional semantics also applies to a more expressive probabilistic language, with loops and higher-order functions.

Narayanan et al. [2016] and Zinkov and Shan [2016] use measure-theoretic denotational semantics to validate the correctness of program transformations in Hakaru, including its programmable Metropolis-Hastings transformation. Like ours, Hakaru's Metropolis-Hastings construct accepts a model program and a kernel program, and automatically produces a transition kernel that satisfies detailed balance, via application of the MH accept-reject rule. Hakaru uses symbolic disintegration and (if necessary) computer algebra to compute the marginal densities it requires for calculating the MH accept/reject ratio. Our approach, by contrast, follows Gen [Cusumano-Towner et al. 2019] and Pyro [Bingham et al. 2019] in interpreting programs as denoting joint distributions over traces of all random variables in a model, so that implementing Metropolis-Hastings (and other programmable inference algorithms) is possible without computing marginal densities. Although a full comparison of symbolic disintegration and trace representations for tractable density calculation is beyond the scope of this paper, we note that there are important differences in the expressiveness of Hakaru and STPL, the language we present here. For example, in Hakaru, users can constrain the results of deterministic functions of random choices, rather than just the primitive random choices themselves, as in STPL. On the other hand, STPL's modeling language supports higher-order functions and $\mathbf{while}_I$ loops with stochastic bodies and termination conditions, whereas Hakaru supports only first-order functions and a **plate** operator for sampling random vectors with conditionally independent entries. Better characterizing the relationship between these two approaches, and the ways in which they might inform one another, is interesting future work.

Concurrently with this work, Lee et al. [2019] have developed a detailed analysis of the semantics of stochastic variational inference with custom variational families that sample from **normal** distributions. They also introduce an abstract interpretation that can verify *model-guide support match*, a similar condition to the one we check, for a subset of Pyro programs; their analysis is in some ways more limited than ours (it does not handle user-defined functions, for example), but does reason about tensor shape and broadcasting (which we have not considered here). We are excited to better understand the differences between trace types and their technique, and between STPL and the subset of Pyro their analysis handles. Ideas from that work could potentially be combined with ours to develop abstract interpretation analyses for Monte Carlo inference programs, or to develop a semantics for variational inference with custom approximating families in quasi-Borel spaces.

Type systems for inference in probabilistic programming languages. The idea of using type systems to establish soundness properties of inference code is well-established. Bhat et al. [2012], for example, use types to ensure that density calculation can be soundly automated, in a class of programs without loops in probabilistic code or branches that can alter the support of a program's output distribution. That work (and other approaches to exact density computation) could be integrated with ours: terms in their language could be directly used as density-carrying measures in STPL. Shuffle [Atkinson et al. 2018] features a type system for verifying hand-coded Monte Carlo inference algorithms, but its focus is not on algorithms that use custom proposal distributions, but rather algorithms that explicitly manipulate densities, or exploit analytic relationships in a model to marginalize out nuisance variables. Shuffle, and earlier systems like BLAISE [Bonawitz 2008] and AugurV2 [Huang et al. 2017], have introduced sound combinators for MCMC kernels, much like the ones we present here. The novelty in our work on kernel combinators comes from two places: (1) our kernels track, in their types, the class of models for which they are stationary, and (2) we introduce a combinator, $\mathbf{if}_{\mathcal{K}}$, for conditional execution, which is only guaranteed to be sound because the type ensures that the condition for execution cannot be negated by the kernel being

executed. Zhou et al. [2019] develop an approach for automatically detecting discontinuities in the density functions for probabilistic programs. Extending our work to use a variant of their algorithm could allow us to formulate more interesting soundness conditions for variational inference, which typically relies on differentiation of objective functions involving densities. Finally, although they do not consider the validation of programmable inference algorithms, Ścibior and Thomas [2019] do suggest using records as strongly-typed traces for straight-line probabilistic programs.

Programmable inference. Programmable inference [Mansinghka et al. 2018] refers to programming constructs that enable user-space customization of inference in models defined by probabilistic programs, to better suit the application at hand. Examples include constructs for custom MCMC schedules [Ge et al. 2018; Mansinghka et al. 2014]; custom Monte Carlo proposals [Bingham et al. 2019; Cusumano-Towner et al. 2019; Murray 2013; Zinkov and Shan 2016]; and custom variational families [Bingham et al. 2019; Cusumano-Towner et al. 2019; Ritchie et al. 2016; Tran et al. 2017]. Probabilistic programs implementing importance sampling proposals and variational families are sometimes referred to as *guide programs*, following Harik and Shazeer [2010]. Inference programming constructs have also been introduced that support custom decompositions of problems into subproblems that can be solved using MCMC, sequential Monte Carlo, variational inference, and optimization [Mansinghka et al. 2018, 2014].

In this paper, we use our type system and semantics to develop sound-by-construction versions of a broad class of programmable inference constructs, but not all. We handle custom scheduling for MCMC, custom proposals for Monte Carlo algorithms, and custom variational families. However, we do not model Venture's stochastic state-dependent subproblem selection [Mansinghka et al. 2014]. Also, we focus on algorithms that leverage program tracing to implement programmable inference, rather than symbolic disintegration and computer algebra [Narayanan et al. 2016; Roberts et al. 2019].

8 DISCUSSION

Our proposed architecture strikes a balance: by forfeiting some flexibility in the modeling language (see Section 5.3), we gain the ability to check soundness properties statically for a broad class of programmable inference algorithms, including custom-proposal variants of importance sampling, sequential Monte Carlo, and MCMC, as well as custom-approximating-family variational inference. The soundness properties we consider are of interest to practitioners because using valid proposals, kernels, and variational families can help to ensure that an inference algorithm produces reasonable results, and does not contain certain kinds of hard-to-isolate bugs.

But these properties are only part of the puzzle: they are often necessary but not sufficient conditions for what programmers care about in practice. We verify that custom-proposal importance sampling and sequential Monte Carlo algorithms yield properly weighted samplers, but not that the variance of the importance weights is bounded, a property that can be used to ensure that finite-sample algorithms will produce reasonable results [Chatterjee and Diaconis 2018; Cortes et al. 2010]. We ensure that transition kernels are stationary, but not that they are ergodic; thus, although our kernels can be used soundly within sequential Monte Carlo or annealed importance sampling algorithms, simply iterating a well-typed kernel indefinitely may fail to sample from the target measure [Andrieu et al. 2003]. And there are many other things one might like to know about a proposal, transition kernel, or variational family, apart from the absolute continuity properties we establish. For example, one might wish to bound the KL divergence between the overall inference algorithm (run with a finite number of Monte Carlo samples or gradient steps) and the target distribution, or bound the expected loss of an application-dependent functional metric.

Building tools that help programmers reason about these bigger questions is an intriguing challenge, touching on fundamental questions in probability and statistics [Diaconis 2009; Freer et al. 2010]. Recent research on program logics for probabilistic programming [Sato et al. 2019] may prove useful in this effort, as these logics allow us to state and reason about not just equivalence of measures but also convergence properties and bounds for distributions described as programs. Another approach would be to develop dynamic analyses of the KL divergence between an inference program and its target distribution, based on recently introduced Monte Carlo estimators [Cusumano-Towner and Mansinghka 2017].

Although we have focused in this paper on how trace types for probabilistic programs can be used to ensure soundness properties of inference algorithms, reasoning about the possible execution traces of a probabilistic program could be useful for other purposes, too. For example, static knowledge about the shapes of execution traces, and the addresses modified by MCMC kernels, could also be used to improve the *performance* of inference, by compiling trace data structures specialized for a particular inference algorithm's needs [Cusumano-Towner and Mansinghka 2018].

Programmers, and programming tools, often rely on the type signature of a function to reason about its behavior without a full characterization of its semantics. Trace types can play an analogous role, helping probabilistic programmers and probabilistic programming tools reason about the behavior of probabilistic programs. Trace types could also potentially facilitate the reuse of modeling and inference code, by helping programmers detect valid opportunities to use components of generative models, proposals, transition kernels, and variational families in contexts for which they were not originally designed. As probabilistic programs continue to scale from tens to hundreds or thousands of lines, and as they begin to be written by groups of probabilistic programmers (not just individuals), it will become increasingly useful to have concise summaries of the semantics of probabilistic code. We hope the concept of trace types, and the semantic framework developed in this paper, helps the probabilistic programming community develop new tools and ways of thinking for reasoning correctly about probabilistic programs.

ACKNOWLEDGMENTS

This material is based upon work supported by philanthropic gifts from the Siegel Family Foundation and from the Aphorism Foundation, and also by a research contract from the Intel Probabilistic Computing Center. We are grateful to the referees for their thoughtful and constructive input, and also to Hengchu Zhang, Jonathan Rees, Cameron Freer, Eric Atkinson, Feras Saad, and Ben Zinberg for useful discussions and suggestions.

REFERENCES

Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An Introduction to MCMC for Machine Learning. *Machine Learning* 50, 1 (Jan 2003), 5–43.

Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying Handcoded Probabilistic Inference Procedures. (May 2018). arXiv:1805.01863

Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A Type Theory for Probability Density Functions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*. ACM, New York, NY, USA, 545–556.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20, Article 28 (Feb. 2019), 6 pages.

Keith A. Bonawitz. 2008. *Composable Probabilistic Inference with Blaise.* Ph.D. Dissertation. Massachusetts Institute of Technology.

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. *J. Stat. Softw.* 76, 1 (2017), 1–32.

Simon Castellan and Hugo Paquet. 2019. Probabilistic Programming Inference Via Intensional Semantics. In *European Symposium on Programming (ESOP 2019) (Lecture Notes in Computer Science)*, Vol. 11423. Springer, Berlin, 322–349.

Joseph T. Chang and David Pollard. 1997. Conditioning as Disintegration. *Stat. Neer.* 51, 3 (Nov. 1997), 287–317.

Sourav Chatterjee and Persi Diaconis. 2018. The Sample Size Required in Importance Sampling. *The Annals of Applied Probability* 28, 2 (2018), 1099–1135.

Corinna Cortes, Yishay Mansour, and Mehryar Mohri. 2010. Learning Bounds for Importance Weighting. In *Advances in Neural Information Processing Systems 23 (NIPS 2010)*. Curran Associates, Inc., Red Hook, NY, USA, 442–450.

Marco Cusumano-Towner and Vikash K. Mansinghka. 2018. A Design Proposal for Gen: Probabilistic Programming with Fast Custom Inference via Code Generation. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, New York, NY, USA, 52–57.

Marco F. Cusumano-Towner and Vikash K. Mansinghka. 2017. AIDE: An Algorithm for Measuring the Accuracy of Probabilistic Inference Algorithms. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. Curran Associates, Inc., Red Hook, NY, USA, 3000–3010.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 221–236.

Persi Diaconis. 2009. The Markov Chain Monte Carlo Revolution. *Bull. Am. Math. Soc* 46, 2 (2009), 179–205.

Adam Foster, Martin Jankowiak, Eli Bingham, Yee Whye Teh, Tom Rainforth, and Noah Goodman. 2019. Variational Optimal Experiment Design: Efficient Automation of Adaptive Experiments. (March 2019). arXiv:1903.05480 NeurIPS 2019 Bayesian Deep Learning Workshop.

Cameron E. Freer, Vikash K. Mansinghka, and Daniel M. Roy. 2010. When are Probabilistic Programs Probably Computationally Tractable?. In *Workshop on Monte Carlo Methods for Modern Applications (NIPS 2010)*. Curran Associates, Inc., Red Hook, NY, USA. http://danroy.org/papers/FreerManRoy-NIPSMC-2010.pdf

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures (FOSSACS 2016) (Lecture Notes in Computer Science)*, Bart Jacobs and Christof Löding (Eds.), Vol. 9634. Springer, Berlin, 513–530.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS 2018) (Proceedings of Machine Learning Research)*, Vol. 84. PMLR, 1682–1690.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis* (3 ed.). Taylor & Francis.

Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2008)*. AUAI Press, 220–229.

Peter J. Green. 1995. Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika* 82, 4 (Dec. 1995), 711–732.

Georges Harik and Noam Shazeer. 2010. Variational Program Inference. (June 2010). arXiv:1006.0991

Chris Heunen, Ohad Kammar, Sam Staton, Sean Moss, Matthijs Vákár, Adam Ścibior, and Hongseok Yang. 2018. The Semantic Structure of Quasi-Borel Spaces. https://pps2018.sice.indiana.edu/files/2018/01/pps18-qbs-semantic-structure.pdf Workshop on Probabilistic Programming Semantics (PPS 2018).

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-order Probability Theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017)*. IEEE, Piscataway, NJ, USA, Article 77, 12 pages.

Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms For Probabilistic Modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 111–125.

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM, New York, NY, USA, 633–645.

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

Jun S. Liu and Rong Chen. 1998. Sequential Monte Carlo Methods for Dynamic Systems. *J. Am. Stat. Assoc.* 93, 443 (1998), 1032–1044.

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 603–616.

Vikash K. Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. (April 2014). arXiv:1404.0099

Lawrence M. Murray. 2013. Bayesian State-Space Modelling on High-performance Hardware Using Libbi. (June 2013). arXiv:1306.3277

Praveen Narayanan. 2019. *Verifiable And Reusable Conditioning*. Ph.D. Dissertation. Indiana University.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference By Program Transformation In Hakaru (System Description). In *Proceedings of the 13th International Symposium on Functional and Logic Programming (FLOPS 2016) (Lecture Notes in Computer Science)*, Vol. 9613. Springer, New York, NY, USA, 62–79.

Daniel Ritchie, Paul Horsfall, and Noah D. Goodman. 2016. Deep Amortized Inference For Probabilistic Programs. (Oct. 2016). arXiv:1610.05735

David A. Roberts, Marcus Gallagher, and Thomas Taimre. 2019. Reversible Jump Probabilistic Programming. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 2019)*, Vol. 89. PMLR, 634–643.

Stuart Russell and Peter Norvig. 2016. *Artificial Intelligence: A Modern Approach* (4 ed.). Pearson Education Limited.

Feras A. Saad, Marco Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling. *Proc. ACM Program. Lang.* 3, POPL, Article 37 (2019), 29 pages.

Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-order Probabilistic Programs: Reasoning About Approximation, Convergence, Bayesian Inference, and Optimization. *Proc. ACM Program. Lang.* 3, POPL, Article 38 (Jan. 2019), 30 pages.

Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell 2015)*. ACM, New York, NY, USA, 165–176.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. *Proc. ACM Program. Lang.* 2, ICFP, Article 83 (July 2018), 29 pages.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational Validation of Higher-Order Bayesian Inference. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2017), 60.

Adam Ścibior and Michael Thomas. 2019. Strongly Typed Tracing of Probabilistic Programs. https://www.cs.ubc.ca/~ascibior/assets/pdf/lafi19a.pdf

Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for Probabilistic Programming: Higher-order Functions, Continuous Distributions, and Soft Constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*. ACM, New York, NY, USA, 525–534.

Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic Robotics*. MIT Press.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *International Conference on Learning Representations (ICLR 2017)*.

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages via Transformational Compilation. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS 2011) (Proceedings of Machine Learning Research)*, Vol. 15. PMLR, 770–778.

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

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS 2014) (Proceedings of Machine Learning Research)*, Vol. 33. PMLR, 1024–1032.

Cheng Zhang, Judith Butepage, Hedvig Kjellstrom, and Stephan Mandt. 2019. Advances in Variational Inference. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 8 (Aug. 2019), 2008–2026.

Yuan Zhou, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. 2019. LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 2019) (Proceedings of Machine Learning Research)*, Vol. 89. PMLR, 148–157.

Robert Zinkov and Chung-chieh Shan. 2016. Composing Inference Algorithms as Program Transformations. (March 2016). arXiv:1603.01882