Probabilistic Programming with Stochastic Probabilities

ALEXANDER K. LEW, MIT, USA
MATIN GHAVAMIZADEH, MIT, USA
MARTIN C. RINARD, MIT, USA
VIKASH K. MANSINGHKA, MIT, USA

We present a new approach to the design and implementation of probabilistic programming languages (PPLs), based on the idea of *stochastically estimating* the probability density ratios necessary for probabilistic inference. By relaxing the usual PPL design constraint that these densities be computed *exactly*, we are able to eliminate many common restrictions in current PPLs, to deliver a language that, for the first time, simultaneously supports first-class constructs for marginalization and nested inference, unrestricted stochastic control flow, continuous and discrete sampling, and programmable inference with custom proposals. At the heart of our approach is a new technique for compiling these expressive probabilistic programs into randomized algorithms for unbiasedly estimating their densities and density reciprocals. We employ these stochastic probability estimators within modified Monte Carlo inference algorithms that are guaranteed to be sound despite their reliance on inexact estimates of density ratios. We establish the correctness of our compiler using logical relations over the semantics of λ_{SP}, a new core calculus for modeling and inference with stochastic probabilities. We also implement our approach in an open-source extension to Gen, called GenSP, and evaluate it on six challenging inference problems adapted from the modeling and inference literature. We find that: (1) GenSP can automate fast density estimators for programs with very expensive exact densities; (2) convergence of inference is mostly unaffected by the noise from these estimators; and (3) our sound-by-construction estimators are competitive with hand-coded density estimators, incurring only a small constant-factor overhead.

CCS Concepts: • **Mathematics of computing** → *Bayesian computation*; *Statistical software*; • **Software and its engineering** → *Compilers*; *Formal language definitions.*

Additional Key Words and Phrases: probabilistic programming, semantics, approximate computing

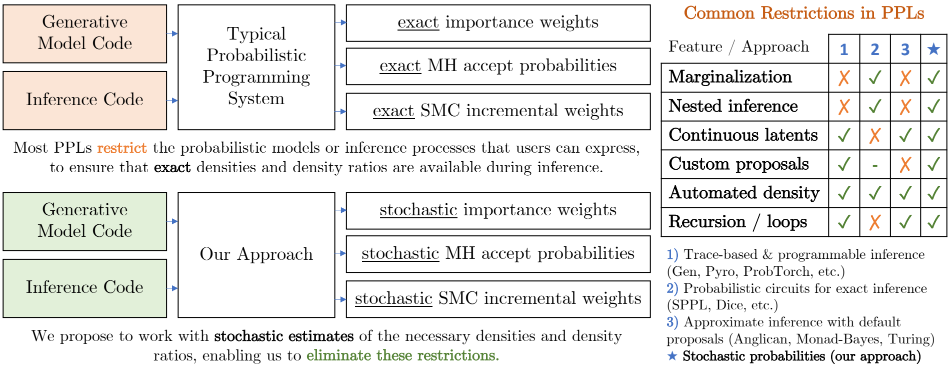
ACM Reference Format:
Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. 2023. Probabilistic Programming with Stochastic Probabilities. *Proc. ACM Program. Lang.* 7, PLDI, Article 176 (June 2023), 25 pages. https://doi.org/10.1145/3591290

1 INTRODUCTION

Probabilistic programming systems have recently emerged as important tools for modeling and inference. They provide practitioners with formal languages for constructing probability distributions, and automated machinery for implementing sound and efficient inference algorithms. These capabilities have helped researchers invent and apply sophisticated modeling and inference techniques, achieving state-of-the-art results in a range of fields, including 3D scene understanding [Gothoskar et al. 2021], time series prediction [Saad et al. 2019], data cleaning [Lew et al. 2021], Bayesian phylogeny [Ronquist et al. 2021], and large-scale scientific simulation [Baydin et al. 2019].

Authors' addresses: Alexander K. Lew, MIT, Cambridge, MA, 02139, USA, alexlew@mit.edu; Matin Ghavamizadeh, MIT, Cambridge, MA, 02139, USA, mghavami@mit.edu; Martin C. Rinard, MIT, Cambridge, MA, 02139, USA, rinard@mit.edu; Vikash K. Mansinghka, MIT, Cambridge, MA, 02139, USA, vkm@mit.edu.

2475-1421/2023/6-ART176
https://doi.org/10.1145/3591290


Generative Model Code
Inference Code
Typical Probabilistic Programming System
exact importance weights
exact MH accept probabilities
exact SMC incremental weights
Most PPLs restrict the probabilistic models or inference processes that users can express, to ensure that exact densities and density ratios are available during inference.
Generative Model Code
Inference Code
Our Approach
stochastic importance weights
stochastic MH accept probabilities
stochastic SMC incremental weights
We propose to work with stochastic estimates of the necessary densities and density ratios, enabling us to eliminate these restrictions.
Common Restrictions in PPLs

Feature / Approach	1	2	3	★
Marginalization	✗	✓	✗	✓
Nested inference	✗	✓	✗	✓
Continuous latents	✓	✗	✓	✓
Custom proposals	✓	-	✗	✓
Automated density	✓	✓	✓	✓
Recursion / loops	✓	✗	✓	✓

1) Trace-based & programmable inference (Gen, Pyro, ProbTorch, etc.)
2) Probabilistic circuits for exact inference (SPPL, Dice, etc.)
3) Approximate inference with default proposals (Anglican, Monad-Bayes, Turing)
★ Stochastic probabilities (our approach)

Fig. 1. Many PPLs place restrictions on the models or inference algorithms users can encode, so exact densities can be efficiently automated. We eliminate these restrictions by using only stochastic estimates.

But the automation that PPLs provide currently comes with a trade-off. To ensure that the *probability density ratios* needed during inference can be automatically and exacty computed, essentially all PPLs place *restrictions* on the models or inference processes that users can express.

Common restrictions in existing probabilistic programming languages. Different languages omit different features, to ensure the tractability of exact densities in inference:

- **Marginalization.** In Gen [Cusumano-Towner et al. 2019], Pyro [Bingham et al. 2019], and ProbTorch [Stites et al. 2021], users' models and proposals must be *joint* probability distributions over many primitive random choices; there is no construct for *marginalizing* variables, because in general, marginal densities would require infinite sums or integrals to compute exactly.
- **Normalization (i.e., nested inference).** With few exceptions, PPLs do not support models and proposals that are *themselves* defined using the posteriors of other probabilistic programs [Rainforth 2018; Zhang and Amin 2022]. Evaluating their densities would require computing *normalizing constants* for the inner models, which are generally not available exactly.
- **Infinite-support latents.** Languages with exact inference, such as Dice [Cheng et al. 2021; Holtzen et al. 2020] and SPPL [Saad et al. 2021], support marginalization and normalization, but forbid or highly restrict the use of *continuous* and other *infinite-support* probability distributions. These restrictions ensure that programs can be compiled to probabilistic circuits with densities that can be computed exactly, as finite sums.[1]
- **Custom proposals.** In Monad-Bayes [Ścibior et al. 2018], Anglican [Wood et al. 2014], and Turing [Ge et al. 2018], the *proposal distributions* used during inference are restricted to match the prior. When combined with another restriction—that *observed variables* must be modeled using primitive distributions as likelihoods—this ensures that the density ratios between models and proposals can be computed exactly, even if their individual densities cannot.
- **Automation for densities.** Some languages, such as Stan [Carpenter et al. 2017], require users to directly encode their models *as* effective procedures for evaluating exact log densities, over a fixed number of continuous random variables.[2]

[1] Some languages with exact inference (e.g., λPSI [Gehr et al. 2020] and Hakaru [Shan and Ramsey 2017]) rely on sound but incomplete computer algebra to simplify some integrals and infinite sums; their modeling languages are not *explicitly* restricted, but *implicit* restrictions arise because on many programs, these symbolic techniques fail to produce densities that can be evaluated. These languages also lack general recursion.

[2] Stan has syntactic sugar that more closely resembles the syntax of other PPLs, and desugaring could be viewed as density automation. But this modeling sub-language has many of the restrictions already mentioned, e.g. *no marginalization.*

This work. We present a new approach to probabilistic programming, based on *stochastic probability estimates* instead of exact densities. This approach lets us support **marginal** and **normalize** as first-class constructs, in the context of a "universal" probabilistic language with discrete and continuous sampling, higher-order functions, and programmable inference. That is, by switching exact densities for stochastic estimates, we can eliminate the restrictions discussed above, simultaneously supporting marginalization, normalization, infinite-support latents, and custom proposals.

Our approach works by compiling these expressive probabilistic programs into randomized algorithms for estimating their probability density functions and density reciprocals. We formalize our compiler as a program transformation on λ_{SP}, a new core calculus for probabilistic programming with stochastic probabilities. We establish the correctness of our approach in two key theorems, showing that our compiler produces unbiased estimators of densities and density reciprocals (Thm. 5.2), and that these unbiased estimators can be soundly employed within custom Monte Carlo inference algorithms (Thm. C.1). We implemented our approach in an open-source tool called GenSP, which extends Gen [Cusumano-Towner et al. 2019] with support for our new constructs. We use GenSP to evaluate our approach on six challenging inference problems, measuring the runtime overhead of our automation (compared to hand-coded versions of the same algorithms), and the impact on convergence of using stochastic probability estimates. In sum, this paper contributes:

(1) The **stochastic probability interface** (Sec. 3), which gives a precise specification for the stochastic density estimation that we propose to automate. The interface is permissive enough to admit a wide range of fast density estimation techniques, but strict enough to ensure that estimates can be used soundly within Monte Carlo algorithms (Thm. C.1).[3]
(2) λ_{SP}, **an expressive core calculus for probabilistic modeling and programmable inference** with discrete and continuous sampling, conditioning, branching, and higher-order functions, plus **first-class constructs for *marginalization* and *approximate normalization*** (Sec. 4).
(3) **Novel program transformations** (Sec. 5) for compiling λ_{SP} programs into implementations of the SPI, and **theoretical validation** of their correctness via semantic logical relations.
(4) GenSP, a **practical open-source implementation** of λ_{SP}, which modifies and extends the Gen probabilistic programming system [Cusumano-Towner 2020] to support λ_{SP}'s new constructs for more expressive modeling and inference (https://github.com/probcomp/GenSP.jl).
(5) An **evaluation of the performance of our approach** (Section 6). In six case studies, we establish that: (1) stochastic probability density estimators automated by GenSP are competitive with hand-coded implementations of the same estimators, and (2) when used within higher-level inference algorithms, the impact of estimator variance on convergence is negligible.

2 EXAMPLE

To introduce our approach, we tour a simplified version of Sec. 6's inverse graphics case study.

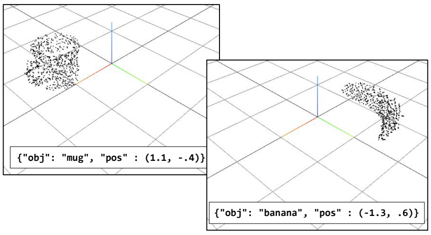
Prior over scene descriptions. Fig. 2 shows λ_{SP} code for a generative model of simple visual scenes. It first defines a *prior distribution* over scene descriptions, by sampling a string-valued object identity `o` uniformly from the list `["mug", "bowl", "banana"]`, and then a 2D position `p` from a Gaussian distribution centered at the origin. Each random choice is made using the **sample** command, which takes as input: (1) a distribution to sample, of type $D\ \sigma$ (where σ is the type of the sampled value), and (2) a string argument, *naming* the sample.

Probabilistic programs like `prior`, which interleave deterministic computation with named **sample** commands, have type $P\ \tau$: they represent *traced* probabilistic computations returning values of type τ. The fact that these computations are *traced* means that they encode *joint probability*

[3] Full paper with appendices available at alexlew.net/papers/stochastic-probabilities.pdf.

Prior over Scenes

```
-- Prior over symbolic scene
-- description (object + position)
prior : P (Str × ℝ^2)
prior = P.do
    let objs = ["mug", "bowl", "banana"]
    o ← sample (uniform objs) "obj"
    p ← sample (normal2d (0,0) 1) "pos"
    return (o, p)
```

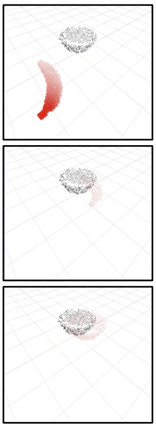

{"obj": "mug", "pos" : (1.1, -.4)}
{"obj": "banana", "pos" : (-1.3, .6)}

Marginalized Likelihood over Depth Images

```
-- Distribution over points in an
-- observed point cloud, given latent cloud
noisy_point_from : [ℝ^3] → D ℝ^3
noisy_point_from = λ cloud.
    marginal
      (P.do pt ← sample (uniform cloud) "p"
            return (normal3d pt 0.1))
      (alg cloud)

-- Model the observed point cloud as iid
-- samples from the noisy point distribution
noisy_render : (Str × ℝ^2) → ℕ → D [ℝ^3]
noisy_render = λ scene. λ n.
    iid n (noisy_point_from
          (render_point_cloud scene))

-- Algorithm for marginalization
alg cloud _ = smc (importance (sample (uniform cloud) "p") 10)
```



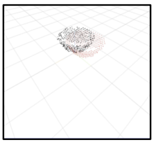
```
estimate_density
  (noisy_render
    ("banana", (-1.5, -0.5))
    1000) observed_cloud
=> -189855

estimate_density
  (noisy_render
    ("banana", (-0.6, 0))
    1000) observed_cloud
=> -12313

estimate_density
  (noisy_render
    ("bowl", (-0.6, 0))
    1000) observed_cloud
=> -2206
```

Unnormalized Posterior

```
-- Unnormalized posterior over scenes,
-- given an observed point cloud
model : [ℝ^3] → M (Str × ℝ^2)
model = λ observed_cloud. M.do
    let n = length observed_cloud
    scene ← prior
    observe (noisy_render scene n) observed_cloud
    return scene
```



```
estimate_density
  (model observed_cloud)
  {"obj": "bowl",
   "pos": (-0.6, 0)}
=> -2210
```

Normalized Posterior

```
-- Metropolis-Hastings proposals
objs = ["mug", "bowl", "banana"]
switch_obj t = sample (uniform objs) "obj"
drift_pos t = sample (normal2d t["pos"] 0.1) "pos"
kernel = sequence (mh switch_obj) (mh drift_pos)
-- Inference: MCMC to infer scene, given point cloud
infer_scene : [ℝ^3] → P (Str × ℝ^3)
infer_scene = λ observed_cloud.
    normalize (model observed_cloud)
        (mcmc prior kernel 50)
```

```
simulate (infer_scene bowl_image)
```

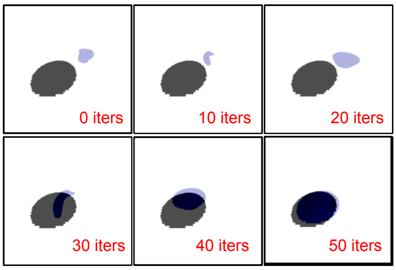


Fig. 2. Overview example: inferring the identity and pose of an object, given a point cloud

distributions over *traces* that record every named sample they make. These traces are finite dictionaries, mapping the names of samples to the values they take on in a particular execution. For example, `{"obj": "mug", "pos": (2.1, 1.1)}` is a possible trace of the program `prior`. The joint density function that `prior` encodes over the variables in its trace can easily be evaluated exactly, as a product of the densities of the primitives used to make each choice: $p_{\texttt{prior}}(\texttt{obj}, \texttt{pos}) = \frac{1}{3} \cdot \mathcal{N}(\texttt{pos}; \mathbf{0}, I)$.

Likelihood over rendered depth images. Given a scene description, what distribution over depth images should we expect to see? Following Gothoskar et al. [2021], we model observed depth images as arising in two steps: first, a clean rendered image is produced using the deterministic `render_point_cloud` method. This returns a list of rendered points in 3D space, based on the scene description. Then, to generate a noisy point cloud from a ground-truth, latent rendered cloud, the likelihood distribution samples the observed points **iid** (independently and identically distributed) from a helper program, `noisy_point_from`. The goal of `noisy_point_from` is to encode the *marginal* distribution that arises when a random point is chosen from the latent cloud, and then perturbed with Gaussian noise. The **marginal** construct allows users to specify such distributions: the user provides a program of type $P\ (D\ \sigma)$ (a probabilistic program that may make many random choices before *returning* a distribution), and **marginal** produces a new "primitive distribution" $D\ \sigma$ that *marginalizes* out all the randomness in the probabilistic program.

Computing this likelihood exactly would require an expensive sum: $p_{\texttt{noisy\_point\_from}}(\mathbf{y} \mid X) = \frac{1}{|X|} \sum_{\mathbf{x} \in X} \mathcal{N}(\mathbf{y}; \mathbf{x}, 0.1I)$. But a key idea in this paper is that exact densities are unnecessary. Distributions instead satisfy a less demanding interface (Sec. 3), requiring only unbiased density estimates. These estimates are themselves generated using inference: any inference algorithm that can be constructed in our DSL can be used to estimate the sums and integrals that arise when marginalizing. As such, as the second argument to **marginal**, users may provide any choice of inference algorithm to estimate the necessary densities. We could, for example, use **enumeration** to perform exact marginalization, at the cost of likelihood evaluation that scales quadratically in the size of the latent and observed point clouds ($p_{\texttt{noisy\_render}}(Y \mid o, \mathbf{p}) = \prod_{\mathbf{y} \in Y} p_{\texttt{noisy\_point\_from}}(\mathbf{y} \mid \texttt{render}(o, \mathbf{p}))$). To reduce cost, here we define a cheaper algorithm (`alg`), enabling linear-time likelihood estimation. If the variance of the estimator is too high, the convergence of our inference will suffer. In this example, a simple importance sampling estimator sufficed, but in the full 3DP3 model [Gothoskar et al. 2021] (Sec. 6), we found we needed a more sophisticated estimator, based on a custom proposal.

Conditioning on data. To combine our prior with our likelihood, we use the **observe** construct, which takes in a likelihood distribution and an observed value. The type $P\ X$ that we saw earlier is reserved for *generative* processes, which encode normalized probability distributions; programs that use **observe** are of type $M\ X$, where M is a monad of *unnormalized* distributions. The program `model` in Fig. 2 implements the unnormalized posterior for our model, generating a scene from the prior and then conditioning on an observed point cloud. Its joint density over traces is

$$p_{\texttt{model(obs)}}(\texttt{obj}, \texttt{pos}) = p_{\texttt{prior}}(\texttt{obj}, \texttt{pos}) \cdot p_{\texttt{noisy\_render}}(\texttt{obs} \mid \texttt{obj}, \texttt{pos}).$$

This density does not integrate to 1, and (like other programs of type $M\ \tau$) it does not represent a probabilistic computation that can be directly run. However, we can still ask for the (estimated) density of a trace. In Fig. 2, note how this allows us to distinguish between good explanations of our observations (which have higher density) and poor explanations (which have lower density).

Inference. To generate plausible explanations for our data, we need to perform *posterior inference*, by *normalizing* our unnormalized model. The `infer_scene` function in Fig. 2 implements a simple inference algorithm that uses **mcmc** to run a 50-step Metropolis-Hastings chain, initialized from the prior. Note that the proposal kernels for MH are themselves specified as programs. The result of **normalize** is a program of type $P\ (\mathrm{Str} \times \mathbb{R}^2)$, which can be simulated to generate samples from the approximate posterior. Like **marginal**, **normalize** is a first-class construct, and as such, the approximate posterior's density can also be estimated unbiasedly and automatically. Importantly, the density we estimate is not that of the exact posterior, but rather of the MCMC approximation:

$$p_{\texttt{infer\_scene(obs)}}(\texttt{obj},\texttt{pos}) = \sum_{o_{0:49}\in\texttt{objs}^{50}} \int_{\mathbb{R}^{2^{50}}} p_{\texttt{prior}}(o_0,p_0)\left(\prod_{t=1}^{49} k(o_t,p_t \mid o_{t-1},p_{t-1})\right) k(\texttt{obj},\texttt{pos} \mid o_{49},p_{49})dp_{0:49},$$

where k composes two MH moves for $p_{\texttt{model(obs)}}$, with proposals $p_{\texttt{switch\_obs}}$ and $p_{\texttt{drift\_pos}}$.

3 THE STOCHASTIC PROBABILITY INTERFACE

In this section, we give precise specifications for two core probability estimation operations, which form the *stochastic probability interface*. In our design, primitive distributions must be manually equipped with implementations of the SPI; Sec. 5 then shows how to automate the SPI for compound programs. The SPI is designed to be as general as possible, admitting many fast density estimation strategies, while still ensuring soundness of inference algorithms that use estimated densities, e.g. to compute sequential Monte Carlo weights or Metropolis-Hastings acceptance probabilities. The SPI is based on *positive unbiased density estimation*:

Definition 3.1 (positive unbiased density estimator). *Let μ and ν be measures on X. A positive unbiased density estimator for μ w.r.t. ν is a probability kernel $\xi : X \to \overline{\mathbb{R}}_{\geq 0}$ such that:*

- *the map $\lambda x.\mathbb{E}_{w\sim\xi(x,\cdot)}[w]$ is a density (i.e., Radon-Nikodym derivative) of μ w.r.t. ν; and*
- *for μ-almost-all $x \in X$, $\mathbb{P}_{w\sim\xi(x,\cdot)}[w > 0] = 1$.*

Although unbiasedness is a very nice property, unbiased density estimators alone are not sufficient to soundly implement many inference algorithms. For example, consider *importance sampling* with a proposal $Q(dx)$ for a target $P(dx)$. The algorithm requires that we sample $x \sim Q$ and then compute the importance weight $w = p(x)/q(x)$, where p and q are densities of P and Q respectively. If we can estimate w unbiasedly, it is still a valid importance weight [Chopin et al. 2020, Chapter 8]. However, separately estimating p and q unbiasedly does not let us estimate their ratio: the ratio of two unbiased estimators is not in general an unbiased estimator for the ratio. This issue also arises in other methods that divide by proposal densities, e.g. Metropolis-Hastings and sequential Monte Carlo. To address this problem, we introduce a second type of density computation:

Definition 3.2 (unbiased density sampler). *Let μ be a probability measure on a measurable space X, and ν a reference measure. Further suppose the probability kernel $\xi : X \to \overline{\mathbb{R}}_{\geq 0}$ is a positive unbiased density estimator for μ with respect to ν. Then the unbiased density sampler corresponding to ξ is the measure $\chi(dx, dw)$ on $X \times \overline{\mathbb{R}}_{\geq 0}$ with density $\lambda(x, w).w$ with respect to $\nu(dx)\xi(x, dw)$.*

We call χ a *sampler* because it is always a probability measure, and sampling a pair (x, w) from χ is equivalent to sampling $x \sim \mu$ and then producing a particular estimate of μ's density that can safely be used in the denominators of density ratios, as the following proposition establishes:

Proposition 3.1. *Suppose μ is a probability measure on a measurable space X, and ν a reference measure. Further suppose ξ is a positive unbiased density estimator of μ w.r.t. ν and that χ is its corresponding unbiased density sampler. Then χ is a probability measure whose first marginal, $\pi_{1*}\chi$, is equal to μ. Further, for any measurable $f : X \to \mathbb{R}_{\geq 0}$, $\mathbb{E}_{(x,w)\sim\chi}\left[\frac{1}{w} \cdot f(x)\right] = \int_X f(x)\nu(dx)$.*

This implies that dividing by the weight w returned from χ successfully divides out μ's density, i.e., that $\mathbb{E}_\chi\left[\frac{1}{w} \mid x\right] = \frac{1}{\frac{d\mu}{d\nu}(x)}$. Indeed, we will use unbiased density samplers to correct the biases of *proposal distributions* in importance sampling, Metropolis-Hastings, and sequential Monte Carlo, a task typically accomplished by dividing out proposal densities. We now give a couple examples.

Example 3.1 (unbiased density sampler for a coin). *Let $\mu = \boldsymbol{bern}(0.5)$ model a fair coin with outcomes $\{\mathtt{t}, \mathtt{f}\}$. Then $\xi(x, dw) = \frac{1}{2} \cdot \delta_{0.25} + \frac{1}{2} \cdot \delta_{0.75}$ is a positive unbiased density estimator. Its corresponding unbiased density sampler is $\chi(dx, dw) = \frac{1}{8} \cdot \delta_{(\mathtt{t},0.25)} + \frac{3}{8} \cdot \delta_{(\mathtt{t},0.75)} + \frac{1}{8} \cdot \delta_{(\mathtt{f},0.25)} + \frac{3}{8} \cdot \delta_{(\mathtt{f},0.75)}$.*

Example 3.2 (unbiased densities from importance sampling). *Let X and Z be measurable spaces, and suppose μ is the first marginal of a joint distribution μ_{joint} over $X \times Z$. We can then estimate the density of μ (w.r.t. a reference measure ν on X) using importance sampling. Let $Q(x, dz)$ be a proposal distribution for importance sampling, with $\mu_{joint} \ll \mu(dx) \cdot Q(x, dz)$. Then the importance sampling estimator $\xi(x, dw) = \int_Z Q(x, dz) \cdot \delta_{\frac{d\mu_{joint}}{d(\nu \otimes Q)}(x,z)}(dw)$ is an unbiased density estimator for μ w.r.t. ν. If additionally $\mu(dx) \cdot Q(x, dz) \ll \mu_{joint}$, then ξ is a positive unbiased density estimator, and its corresponding unbiased density sampler is $\chi(dx, dw) = \int_Z \mu_{joint}(dx, dz) \cdot \delta_{\frac{d\mu_{joint}}{d(\nu \otimes Q)}(x,z)}(dw)$.*

Interface. Our *stochastic probability interface* comprises two methods, for a measure μ:

- `estimate_density`: Given x, return $w \sim \xi(x, dw)$ for some unbiased density estimator ξ for μ.
- `simulate`: Return $(x, w) \sim \chi$, the unbiased density sampler corresponding to ξ.

The specification for `simulate` only makes sense when μ is a probability measure. In this case we say that the *full stochastic probability interface* is implemented by ξ and χ. In the case where μ is not a probability measure, we implement only the *restricted stochastic probability interface*, which includes just the `estimate_density` method. Appx. C shows how these operations can be used soundly within popular Monte Carlo algorithm templates.

4 SYNTAX AND SEMANTICS

We formalize our approach on λ_{SP}, a new core calculus for modeling and inference (Fig. 3).

Ground types: As ground types, λ_{SP} features a unit type 1, Booleans $\mathbb{B}$, non-negative extended reals $\overline{\mathbb{R}}_{\geq 0}$ (used to represent importance weights and densities), natural numbers $\mathbb{N}$, real vectors $\mathbb{R}^n$, strings Str, and a type Trace of *traces*, which are finite dictionaries mapping string-valued *names* to (heterogeneous) values of ground type. The term $\{\}$ represents the empty trace, and the syntax $\{t_1 \mapsto t_2\}$ builds a singleton trace mapping the name t_1 to the value t_2. In Fig. 3, c ranges over *constants* of all types, including ground-type constants like **true** and 3.2, as well as primitive functions. We assume primitive functions for manipulating traces, including $concat$: Trace $\longrightarrow$ Trace $\longrightarrow$ Trace (which returns $\{\}$ if names overlap in the two input traces) and $lookup_\sigma$: Str $\longrightarrow$ Trace $\longrightarrow \sigma$ (which returns a default value if the name does not correspond to a value of type σ in the trace). As sugar for concatenation and lookups, we write $t_1 +\!\!+ \, t_2$ and $t_1[t_2]$ respectively.

Probabilistic types: For each ground type σ, there is a type $D\,\sigma$ of *distributions* over σ. For example, the expression **normal** t_1 t_2 has type $D\,\mathbb{R}$ and represents a Gaussian distribution with a user-specified mean and standard deviation. For *every* type τ (including at higher-order), there are types $P\ \tau$ and $M\ \tau$ of traced probabilistic programs and traced measure programs, respectively. P and M are monads: the types $P\ \tau$ and $M\ \tau$ represent a certain kind of effectful computation returning values of type τ. One intuition, which we will make more precise in our semantics, is that these monadic programs interleave arbitrary deterministic computation with named random samples from *distributions* $D\ \sigma$ over ground types. As such, each term of type $P\ \tau$ or $M\ \tau$ encodes (1) a joint distribution over *traces*, which map the names of sampling statements to ground-type values sampled in a particular execution, as well as (2) a deterministic map from traces to values of type τ, encoding the program's output as a function of the random choices it makes. These traced programs can be constructed using the terms: **return** t (which makes no samples and deterministically computes the expression t); **sample** t_d t (which samples a single random variable with a user-provided *name* t : Str from a distribution $t_d : D\,\sigma$); and **do**$\{x \leftarrow t; m\}$ (which first runs t to stochastically

Deterministic Core

$$\begin{aligned}
\text{Ground types } \sigma &::= 1 \mid \mathbb{B} \mid \overline{\mathbb{R}}_{\geq 0} \mid \mathbb{R}^n \mid \mathbb{N} \mid \text{Str} \mid \text{Trace} \\
\text{Types } \tau &::= \sigma \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
\text{Terms } t &::= () \mid c \mid x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \pi_1\, t \mid \pi_2\, t \mid \textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \mid \{\} \mid \{t_1 \mapsto t_2\} \mid t_1 +\!\!\!+\, t_2 \mid t_1[t_2]
\end{aligned}$$

$$\frac{}{\{\} : \text{Trace}} \qquad \frac{t_1 : \text{Str} \quad t_2 : \sigma}{\{t_1 \mapsto t_2\} : \text{Trace}} \qquad \frac{t_1 : \text{Trace} \quad t_2 : \text{Trace}}{t_1 +\!\!\!+\, t_2 : \text{Trace}} \qquad \frac{t_1 : \text{Trace} \quad t_2 : \text{Str}}{t_1[t_2] : \sigma}$$

$\textbf{let } x = t_1 \textbf{ in } t_2$ is sugar for $(\lambda x.t_2)\ t_1$

Probabilistic Programming with Named Samples

$$\begin{aligned}
\text{Types } \tau_p &::= D\ \sigma \text{ (distributions)} \mid P\ \tau \text{ (probabilistic programs)} \mid M\ \tau \text{ (unnormalized programs)} \\
\text{Distribution terms } t_d &::= \textbf{normal}\, t_1\, t_2 \mid \textbf{bernoulli}\, t \mid \textbf{uniform}\, t_1\, t_2 \mid \ldots \\
\text{Probabilistic program terms } t_p &::= \textbf{return}\, t \mid \textbf{sample}\, t_d\, t \mid \textbf{observe}\, t_d\, t \mid P.\textbf{do}\{m\} \mid M.\textbf{do}\{m\} \\
\textbf{do} \text{ notation } m &::= t_p \mid x \leftarrow t_p; m
\end{aligned}$$

$$\frac{t : \tau}{\textbf{return}\, t : P\,\tau} \qquad \frac{t_d : D\,\sigma \quad t : \text{Str}}{\textbf{sample}\, t_d\, t : P\,\sigma} \qquad \frac{t_d : D\,\sigma \quad t : \sigma}{\textbf{observe}\, t_d\, t : M\,1}$$

$$\frac{t_p : P\,\tau}{t_p : M\,\tau} \qquad \frac{t_p : P\,\tau}{P.\textbf{do}\{t_p\} : P\,\tau} \qquad \frac{t_p : M\,\tau}{M.\textbf{do}\{t_p\} : M\,\tau}$$

$$\frac{t_p : P\,\tau_1 \quad x : \tau_1 \vdash P.\textbf{do}\,\{m\} : P\,\tau_2}{P.\textbf{do}\{x \leftarrow t_p; m\} : P\,\tau_2} \qquad \frac{t_p : M\,\tau_1 \quad x : \tau_1 \vdash M.\textbf{do}\,\{m\} : M\,\tau_2}{M.\textbf{do}\{x \leftarrow t_p; m\} : M\,\tau_2}$$

$\textbf{let } x = t; m$ is sugar for $x \leftarrow \textbf{return}\, t; m$

$t_p; m$ is sugar for $\_ \leftarrow t_p; m$

Monte Carlo Programmable Inference

$$\begin{aligned}
\text{Types } \tau_i &::= \text{Alg} \mid \text{SMC} \mid \text{MCMC} \\
\text{Exact algorithms } t_i &::= \textbf{enumeration} \\
\text{SMC algorithms } t_i &::= \textbf{importance}\, t_p\, t \mid \textbf{step}\, t_i\, t_1\, t_2\, t_3 \mid \textbf{resample}\, t_i\, t_1\, t_2 \mid \textbf{rejuvenate}\, t_i\, t_i \mid \textbf{smc}\, t_i \\
\text{MCMC algorithms } t_i &::= \textbf{mh}\, t \mid \textbf{sequence}\, t_i\, t_i \mid \textbf{mcmc}\, t_p\, t_i\, t \\
\text{Applying inference } t_p &::= \textbf{marginal}\, t_p\, t_i \mid \textbf{normalize}\, t_p\, t_i
\end{aligned}$$

$$\frac{t_p : M\,\tau \quad t_i : \text{Alg}}{\textbf{normalize}\, t_p\, t_i : P\,\tau} \qquad \frac{t_1 : P\,(D\,\sigma) \quad t_2 : \sigma \to \text{Alg}}{\textbf{marginal}\, t_1\, t_2 : D\,\sigma} \qquad \frac{t : \text{SMC}}{\textbf{smc}\, t : \text{Alg}} \qquad \frac{t_p : P\,\tau \quad t_i : \text{MCMC} \quad t : \mathbb{N}}{\textbf{mcmc}\, t_p\, t_i\, t : \text{Alg}}$$

$$\frac{t_p : P\,\tau \quad t : \mathbb{N}}{\textbf{importance}\, t_p\, t : \text{SMC}} \qquad \frac{t_i : \text{SMC} \quad t_1 : \mathbb{R}_{\geq 0} \quad t_2 : \mathbb{N}}{\textbf{resample}\, t_i\, t_1\, t_2 : \text{SMC}} \qquad \frac{t_1 : \text{SMC} \quad t_2 : \text{MCMC}}{\textbf{rejuvenate}\, t_1\, t_2 : \text{SMC}}$$

$$\frac{t_i : \text{SMC} \quad t_1 : \text{Trace} \to P\,\tau \quad t_2 : \text{Trace} \to P\,\tau \quad t_3 : M\,\tau}{\textbf{step}\, t_i\, t_1\, t_2\, t_3 : \text{SMC}} \qquad \frac{t : \text{Trace} \to P\,\tau}{\textbf{mh}\, t : \text{MCMC}}$$

$$\frac{t_1 : \text{MCMC} \quad t_2 : \text{MCMC}}{\textbf{sequence}\, t_1\, t_2 : \text{MCMC}} \qquad \frac{}{\textbf{enumeration} : \text{Alg}}$$

Fig. 3. Syntax of the λ_{SP} calculus.

generate a value x, and then runs the remainder of the computation $\textbf{do}\{m\}$). Terms of type $M\,\tau$ may also include **observe** $t_d\ t$ statements that condition the program on the observation that t

Semantics of Ground Types as Measure Spaces (X, Σ_X, ν_X)

$$\begin{array}{l|l|l}
\llbracket \mathbb{B} \rrbracket = (\{\mathbf{true}, \mathbf{false}\}, \mathcal{P}(\{\mathbf{true}, \mathbf{false}\}), \#) & \llbracket \overline{\mathbb{R}}_{\geq 0} \rrbracket = (\mathbb{R}_{\geq 0} \cup \{\infty\}, \mathcal{B}(\overline{\mathbb{R}}_{\geq 0}), \overline{\Lambda}) & \llbracket \mathbb{R}^n \rrbracket \ = (\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n), \Lambda^n) \\
\llbracket \mathbb{N} \rrbracket = (\mathbb{N}, \mathcal{P}(\mathbb{N}), \#) & \llbracket \mathrm{Str} \rrbracket \ = (\mathrm{Str}, \mathcal{P}(\mathrm{Str}), \#) & \llbracket \mathrm{Trace} \rrbracket = (\mathbb{T}, \Sigma_{\mathbb{T}}, \nu_{\mathbb{T}})
\end{array}$$

For all ground types σ, $\llbracket \sigma \rrbracket_m = \llbracket \sigma \rrbracket_d = (\pi_1(\llbracket \sigma \rrbracket), M_{\pi_2(\llbracket \sigma \rrbracket)})$, the canonical QBS for $\llbracket \sigma \rrbracket$'s measurable space.

Semantics of Probabilistic and Higher-Order Types as Quasi-Borel Spaces

$$\begin{array}{ll|ll}
\multicolumn{2}{c|}{\llbracket \cdot \rrbracket_m\text{: measure semantics}} & \multicolumn{2}{c}{\llbracket \cdot \rrbracket_d\text{: density estimator semantics } (\llbracket \cdot \rrbracket_d = \llbracket d\{\cdot\} \rrbracket_t)} \\
\llbracket D\ \sigma \rrbracket_m & = \mathrm{Prob}_{\ll \nu_\sigma} \llbracket \sigma \rrbracket_m & \llbracket D\ \sigma \rrbracket_d & = (\llbracket \sigma \rrbracket_m \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}) \times \mathrm{Prob}\ (\llbracket \sigma \rrbracket_m \times \overline{\mathbb{R}}_{\geq 0}) \\
\llbracket P\ \tau \rrbracket_m & = \mathrm{Prob}_{\ll \nu_{\mathbb{T}}} \mathbb{T} \times (\mathbb{T} \Rightarrow \llbracket \tau \rrbracket_m) & \llbracket P\ \tau \rrbracket_d & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ (\overline{\mathbb{R}}_{\geq 0} \times \mathbb{T})) \times \mathrm{Prob}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}) \times (\mathbb{T} \Rightarrow \llbracket \tau \rrbracket_d) \\
\llbracket M\ \tau \rrbracket_m & = \mathrm{Meas}_{\ll \nu_{\mathbb{T}}} \mathbb{T} \times (\mathbb{T} \Rightarrow \llbracket \tau \rrbracket_m) & \llbracket M\ \tau \rrbracket_d & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ (\overline{\mathbb{R}}_{\geq 0} \times \mathbb{T})) \times (\mathbb{T} \Rightarrow \llbracket \tau \rrbracket_d) \\
\llbracket \tau_1 \to \tau_2 \rrbracket_m & = \llbracket \tau_1 \rrbracket_m \times \llbracket \tau_1 \rrbracket_d \Rightarrow \llbracket \tau_2 \rrbracket_m & \llbracket \tau_1 \to \tau_2 \rrbracket_d & = \llbracket \tau_1 \rrbracket_d \Rightarrow \llbracket \tau_2 \rrbracket_d \\
\llbracket \mathrm{Alg} \rrbracket_m & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \mathrm{Prob}_{\ll \nu_{\mathbb{T}}} \mathbb{T} & \llbracket \mathrm{Alg} \rrbracket_d & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \llbracket D\ \mathrm{Trace} \rrbracket_d \\
\llbracket \mathrm{SMC} \rrbracket_m & = \mathrm{Prob}\ (\mathrm{List}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})) & \llbracket \mathrm{SMC} \rrbracket_d & = \llbracket \mathrm{SMC} \rrbracket_m \times (\mathbb{T} \Rightarrow \llbracket \mathrm{SMC} \rrbracket_m \times \mathbb{N}) \times (\mathbb{T} \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \\
\llbracket \mathrm{MCMC} \rrbracket_m & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}) \Rightarrow & \llbracket \mathrm{MCMC} \rrbracket_d & = (\mathbb{T} \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}) \Rightarrow ((\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \mathrm{Prob}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}))^2 \\
 & \quad (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \mathrm{Prob}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}) & &
\end{array}$$

Semantics of Terms

The measure semantics $\llbracket \cdot \rrbracket_m$ interprets $\Gamma \vdash t : \tau$ as a map $\llbracket \Gamma \rrbracket_m \times \llbracket \Gamma \rrbracket_d \to \llbracket \tau \rrbracket_m$.
The density estimator semantics $\llbracket \cdot \rrbracket_d$ yields a map $\llbracket \Gamma \rrbracket_d \to \llbracket \tau \rrbracket_d$. It arises as the composition of Section 5's program transformation $d\{\cdot\}$ with the semantics of the translation's target language ($\llbracket \cdot \rrbracket_d = \llbracket d\{\cdot\} \rrbracket_t$).

Deterministic Core (Selected Examples)

$$\llbracket c \rrbracket_m(\overline{\gamma}) = \underline{c} \ \big|\ \llbracket x \rrbracket_m(\overline{\gamma}) = \overline{\gamma}_m(x) \ \big|\ \llbracket t_1\ t_2 \rrbracket_m(\overline{\gamma}) = \llbracket t_1 \rrbracket_m(\overline{\gamma})(\llbracket t_2 \rrbracket_m(\overline{\gamma}), \llbracket t_2 \rrbracket_d(\overline{\gamma}_d))$$

Primitive Distributions (Selected Examples)

$$\begin{aligned}
\llbracket \mathbf{normal}\ t_1\ t_2 \rrbracket_m(\overline{\gamma}, dx) &= \mathcal{N}(x; \llbracket t_1 \rrbracket_m(\overline{\gamma}), \llbracket t_2 \rrbracket_m(\overline{\gamma})) \cdot \Lambda(dx) \\
\llbracket \mathbf{bernoulli}\ t \rrbracket_m(\overline{\gamma}, dx) &= \llbracket t \rrbracket_m(\overline{\gamma}) \cdot \delta_{\mathbf{true}}(dx) + (1 - \llbracket t \rrbracket_m(\overline{\gamma})) \cdot \delta_{\mathbf{false}}(dx)
\end{aligned}$$

Traced Programs

$$\begin{aligned}
\llbracket \mathbf{return}\ t \rrbracket_m(\overline{\gamma}) &= (\lambda du.\delta_{\{\}}(du), \lambda u.\llbracket t \rrbracket_m(\overline{\gamma})) \\
\llbracket \mathbf{sample}\ t_d\ t \rrbracket_m(\overline{\gamma}) &= (\lambda du.\ \textstyle\int_{\llbracket \sigma \rrbracket_m} \llbracket t_d \rrbracket_m(\overline{\gamma}, dx)\delta_{\{\llbracket t \rrbracket_m(\overline{\gamma}) \mapsto x\}}(du), \lambda u.u[\llbracket t \rrbracket_m(\overline{\gamma})]) \\
\llbracket \mathbf{observe}\ t_d\ t \rrbracket_m(\overline{\gamma}) &= (\lambda du.\tfrac{d\llbracket t_d \rrbracket_m(\overline{\gamma})}{d\nu_\sigma}(\llbracket t \rrbracket_m(\overline{\gamma})) \cdot \delta_{\{\}}(du), \lambda u.()) \\
\llbracket \mathbf{do}\{x \leftarrow t_p; m\} \rrbracket_m(\overline{\gamma}) &= (\lambda du.\ \textstyle\iint_{\mathbb{T} \times \mathbb{T}} (\pi_1 \circ \llbracket t_p \rrbracket_m)(\overline{\gamma}, du_1) \\
&\qquad (\pi_1 \circ \llbracket \mathbf{do}\{m\} \rrbracket_m) \left(\begin{bmatrix} \overline{\gamma}_m[x \mapsto (\pi_2 \circ \llbracket t_p \rrbracket_m)(\overline{\gamma})(u_1)] \\ \overline{\gamma}_d[x \mapsto (\pi_3 \circ \llbracket t_p \rrbracket_d)(\overline{\gamma}_d)(u_1)] \end{bmatrix}, du_2 \right) \\
&\qquad [\mathit{disj}(u_1, u_2)] \cdot \delta_{u_1 + u_2}(du), \\
&\quad \lambda u.(\pi_2 \circ \llbracket \mathbf{do}\{m\} \rrbracket_m) \left(\begin{bmatrix} \overline{\gamma}_m[x \mapsto (\pi_2 \circ \llbracket t_p \rrbracket_m)(\overline{\gamma})(u)] \\ \overline{\gamma}_d[x \mapsto (\pi_3 \circ \llbracket t_p \rrbracket_d)(\overline{\gamma}_d)(u)] \end{bmatrix} \right)(u))
\end{aligned}$$

Inference Programming (Selected Examples)

$$\begin{aligned}
\llbracket \mathbf{marginal}\ t_p\ t \rrbracket_m(\overline{\gamma}, dx) &= \textstyle\int_{\mathbb{T}} (\pi_1 \circ \llbracket t_p \rrbracket_m)(\overline{\gamma}, du)(\pi_2 \circ \llbracket t_p \rrbracket_m)(\overline{\gamma}, u, dx) \\
\llbracket \mathbf{normalize}\ t_p\ t_i \rrbracket_m(\overline{\gamma}) &= (\llbracket t_i \rrbracket_m(\overline{\gamma})(\lambda u.\lambda dw.\ \textstyle\iint_{\overline{\mathbb{R}}_{\geq 0} \times \mathbb{T}} \pi_1(\llbracket t_p \rrbracket_d(\overline{\gamma}_d))(u, dv, du')\delta_{v \cdot [u' = \{\}]}(dw)), \pi_2(\llbracket t_p \rrbracket_m(\overline{\gamma}))) \\
\llbracket \mathbf{mcmc}\ t_p\ t_i\ t \rrbracket_m(\overline{\gamma})(\rho, du) &= \textstyle\iint_{\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}} (\pi_1(\llbracket t_p \rrbracket_m(\overline{\gamma})))(du_1)\rho(u_1, dw_1)\pi_{1*}(\llbracket t_i \rrbracket_m(\overline{\gamma})(\rho)^{\llbracket t \rrbracket_m(\overline{\gamma})}(u_1, w_1))(du) \\
\llbracket \mathbf{smc}\ t \rrbracket_m(\overline{\gamma})(\rho, du) &= \textstyle\int \llbracket t \rrbracket_m(\overline{\gamma}, d\mathbf{u}, d\mathbf{w}, d\mathbf{v}) \int_{\overline{\mathbb{R}}_{\geq 0}^{|\mathbf{u}|}} (\prod_{j=1}^{|\mathbf{u}|} \rho(u_j, dp_j)) \sum_{k=1}^{|\mathbf{u}|} \frac{w_k \cdot p_k / v_k}{\sum_l w_l \cdot p_l / v_l} \delta_{u_k}(du) \\
\llbracket \mathbf{importance}\ t_p\ t \rrbracket_m(\overline{\gamma}) &= \lambda(d\mathbf{u}, d\mathbf{w}, d\mathbf{v}).\ \textstyle\prod_{j=1}^{\llbracket t \rrbracket_m(\overline{\gamma})} (\pi_2 \circ \llbracket t_p \rrbracket_d)(\overline{\gamma}, du_j, dv_j)\ \delta_1(dw_j)
\end{aligned}$$

Fig. 4. Semantics of the λ_{SP} calculus.

was generated from the distribution t_d; as such, they do not describe straightforward generative processes (probability measures), but rather *unnormalized measures*, to which Bayesian inference must be applied to yield posterior probability distributions. (Any $P\,\tau$ can trivially be regarded as an $M\,\tau$, but to go in the other direction, the user must apply the **normalize** construct to the measure, which renormalizes the measure to a probability distribution by performing approximate Bayesian inference.) We call programs of type $P\,\tau$ or $M\,\tau$ *traced* because they encode distributions over *traces* mapping the names of the random variables they sample to values. For example, the program $P.\mathbf{do}\{x \leftarrow \mathbf{sample}\ (\mathbf{normal}\ 0\ 1)\ \texttt{"x"}; \mathbf{sample}\ (\mathbf{normal}\ x\ 1)\ \texttt{"y"}\}$, of type $P\,\mathbb{R}$, generates traces of the form $\{\texttt{"x"} \mapsto x, \texttt{"y"} \mapsto y\}$, where x and y are real numbers.

Semantics of ground types. Our semantics assigns to each ground type σ a *measure space* $\llbracket \sigma \rrbracket = (X_\sigma, \Sigma_{X_\sigma}, \nu_\sigma)$ (Fig. 4, top). The first two components $(X_\sigma, \Sigma_{X_\sigma})$ define a measurable space of values of type σ; the third component ν_σ is a *reference measure* on this space, with respect to which all *density functions* will be computed. Our choices are largely standard: for discrete types 1, $\mathbb{B}$, Str, and $\mathbb{N}$, we choose the discrete σ-algebra, and the counting reference measure #. For $\mathbb{R}^n$, we choose the Borel σ-algebra and the Lebesgue reference measure Λ. The type $\overline{\mathbb{R}}_{\geq 0}$ denotes the non-negative reals extended with a point at ∞; a set is measurable if its intersection with $\mathbb{R}$ is Borel-measurable. The reference measure $\overline{\Lambda}$ assigns measure $\Lambda(E \cap \mathbb{R})$ to any measurable set E. We must be a bit more careful when defining the measure space $(\mathbb{T}, \Sigma_{\mathbb{T}}, \nu_{\text{Trace}})$ of traces, which may store other traces as values—we defer the definition of this measure space to Appendix A.

Semantics of probabilistic and higher-order types. To reason about higher-order probabilistic programs, we use not measurable spaces but *quasi-Borel spaces* [Heunen et al. 2017], a drop-in replacement for measurable spaces suitable for carrying out measure theory, but with well-behaved function spaces. We refer the reader to Ścibior et al. [2018] for an overview of the use of quasi-Borel spaces to reason about probabilistic programming, or to Appendix B for a brief introduction.

The semantic function $\llbracket \cdot \rrbracket_m$ in Fig. 3 gives the *measure semantics* of λ_{SP}. It interprets ground types σ as quasi-Borel spaces $(X_\sigma, M_{\Sigma_\sigma})$ and distributions $D\,\sigma$ as quasi-Borel probability measures on $\llbracket \sigma \rrbracket_m$ (absolutely continuous w.r.t. the reference measure ν_σ). To interpret traced programs $P\,\tau$ and $M\,\tau$, we define monads P and M on the category of quasi-Borel spaces. For any quasi-Borel space X, $P\,X$ is the product space $\text{Prob}_{\ll \nu_{\text{Trace}}} \mathbb{T} \times (\mathbb{T} \Rightarrow X)$, and $M\,X$ is the product space $\text{Meas}_{\ll \nu_{\text{Trace}}} \mathbb{T} \times (\mathbb{T} \Rightarrow X)$. That is, we interpret traced probabilistic computations as pairs, combining a measure on *traces* (absolutely continuous with respect to ν_{Trace}) with a "value function" from traces to the quasi-Borel space X. To lift a pure computation $x : X$ into the monad P or M, we construct the Dirac measure $\delta_{\{\}}$ on empty traces, and pair it with the value function $\lambda\_.x$. The multiplication of the monad is somewhat more involved, and is defined in Fig. 4's semantics of **do**.

Because distributions of type $D\,\sigma$ denote measures *absolutely continuous with respect to* ν_σ, and traced programs denote measures *absolutely continuous with respect to* ν_{Trace}, we can talk about the *densities* of our probabilistic terms. In Section 5, we will develop a program transformation $d\{\cdot\}$ that translates programs into new programs that implement unbiased density estimators for the measures that the original programs denote. Ideally, we would avoid discussing densities and their estimators *until* Section 5, focusing in this section only on the specification of the λ_{SP}'s semantics. But λ_{SP} makes it possible to *observe* the density estimator that $d\{\cdot\}$ produces, inside the language: we can write a λ_{SP} program, e.g., that runs importance sampling using automatically estimated target or proposal densities. In order to define the semantics of such a program, we need to know precisely what density estimator the importance sampler uses.

In order to address this difficulty, we define an auxiliary semantic function $\llbracket \cdot \rrbracket_d$, with the property that $\llbracket \tau \rrbracket_d = \llbracket d\{\tau\} \rrbracket_m$. We call $\llbracket \cdot \rrbracket_d$ the *density estimator semantics*: it gives the mathematical object denoted by the *translation* of a program *into* a density estimator for the program. With $\llbracket \cdot \rrbracket_d$ in hand, we can give a compositional measure semantics of terms: for a term t of type τ in context Γ, $[\![\Gamma \vdash t : \tau]\!]_m$ is a function $[\![\Gamma]\!]_m \times [\![\Gamma]\!]_d \to [\![\tau]\!]_m$, mapping *pairs* of environments—the first giving *values* of free variables, and the second giving *density estimators* for them—to values in $[\![\tau]\!]_m$. This reflects the fact that if the environment contains a variable $x : D\,\sigma$, e.g., the term t's semantics might depend on the particular density estimator compiled for x. This dependence on $[\![\cdot]\!]_d$ also appears in the measure semantics of function types: we have $[\![\tau_1 \to \tau_2]\!]_m = [\![\tau_1]\!]_m \times [\![\tau_1]\!]_d \Rightarrow [\![\tau_2]\!]_m$.

The bottom part of Fig. 4 gives the measure semantics of terms, where we write $\overline{\gamma} = (\overline{\gamma}_m, \overline{\gamma}_d)$ for a pair of environments. To help readers parse the definitions, which are largely standard for traced probabilistic programming (see, e.g., Lew et al. [2020]), we consider an illustrative example, the semantics of $P.\mathbf{do}\{x \leftarrow t_p; m\}$, where $t_p : P\,\tau_1$ and $x : \tau \vdash P.\mathbf{do}\{m\} : P\,\tau_2$. Given $\overline{\gamma}$, the semantics outputs a pair, where the first component is a measure over traces and the second is a function from traces to $[\![\tau_2]\!]_m$. For the measure, we first draw u_1 from $(\pi_1 \circ [\![t_p]\!]_m)(\overline{\gamma})$, the measure on traces that t_p denotes. We then draw u_2 from the trace measure denoted by $\mathbf{do}\{m\}$, but in an extended $\overline{\gamma}$. The extended environment is obtained by adding a binding for x to each of $\overline{\gamma}_m$ and $\overline{\gamma}_d$. These bindings should have type $[\![\tau_1]\!]_m$ and $[\![\tau_1]\!]_d$, respectively; the value for x in each case comes from the *return value function* that t_p denotes ($\pi_2 \circ [\![t_p]\!]_m$, or $\pi_3 \circ [\![t_p]\!]_d$), applied to u_1. Finally, we concatenate the traces u_1 and u_2 to return the output trace u. The second output of the interpretation, the value function, is simpler: given an input trace u, it applies $[\![t_p]\!]_m$'s value function to u, then applies $[\![\mathbf{do}\{m\}]\!]_m$'s value function (in an extended environment, as above) to u.[4]

Syntax and semantics of inference. Finally, Figs. 3 and 4 give syntax and semantics for λ_{SP}'s inference constructs, which build inference algorithms of type Alg. These can be *used* in two ways:

- *Approximate normalization*: Given an unnormalized program $p : M\,\tau$, and an algorithm a : Alg, **normalize** p a has type $P\,\tau$. It is the program that *runs* the inference algorithm to generate a trace from the normalized posterior of p, and then returns the value of p on that trace.
- *Marginalization*: Consider a program $p : P\,(D\,\sigma)$. The *marginal distribution* of p is the distribution over $[\![\sigma]\!]_m$ that arises by first generating a trace u of p, computing the value d of p on u, and then generating and returning $x \sim d$. The **marginal** construct accepts p as input and outputs a value of type $D\,\sigma$, representing its marginal distribution. It *also* accepts a value-dependent choice of inference algorithm $a : \sigma \to$ Alg. This does *not* affect the term's measure semantics, but will affect the density estimator that the resulting $D\,\sigma$ uses. The goal of a is to, given a value $x \in [\![\sigma]\!]_m$, infer a trace u of p that makes x likely. In Section 5, we will see how having such an inference algorithm helps us estimate the marginal density of x; furthermore, the better that a is at inferring u given x, the lower the variance of the resulting density estimator.

In our semantics, $[\![\text{Alg}]\!]_m$ is the space of functions that take in a target density estimator, and output a distribution over traces; intuitively, the output distribution should approximate the normalized version of the target measure. An Alg can be created using **mcmc** or **smc**, or **enumeration**. The **mcmc** construct accepts an initial distribution, a description of an MH algorithm (constructed using **sequence** and **mh**), and a number of steps, and returns an Alg that initializes a trace from the initial proposal, runs the MH algorithm for a given number of steps, and returns the resulting trace. The **smc** construct accepts a description of an SMC algorithm (constructed using **importance** to initialize a particle collection, **step** to take SMC steps, **rejuvenate** to perform MCMC rejuvenation, and **resample** to do a resampling step), and returns an Alg that runs the SMC algorithm, samples a trace from the resulting weighted particle collection, and returns that trace.

[4] It may seem incorrect to pass the full trace u (which concatenates traces from each subterm) to the value functions for each subterm. But this is OK, thanks to a feature of the value functions in our semantics: when $u \sim (\pi_1 \circ [\![t_p]\!]_m)(\overline{\gamma})$, with probability 1, $(\pi_2 \circ [\![t_p]\!]_m)(\overline{\gamma})(u +\!\!\!+ u') = (\pi_2 \circ [\![t_p]\!]_m)(\overline{\gamma})(u)$ for all u' with names disjoint from those in u. That is, if u is a valid trace of t_p, then we can concatenate extra values to u without changing the value of t_p's value function on u.

$$\text{Types:}\quad [\![\sigma]\!]_t = [\![\sigma]\!]_m \;\Big|\; [\![\tau_1 \times \tau_2]\!]_t = [\![\tau_1]\!]_t \times [\![\tau_2]\!]_t \;\Big|\; [\![\tau_1 \to \tau_2]\!]_t = [\![\tau_1]\!]_t \Rightarrow [\![\tau_2]\!]_t \;\Big|\; [\![T\ \tau]\!] = \text{Prob}\ [\![\tau]\!]_t$$

$$\text{Deterministic terms (selected examples):}\quad [\![c]\!]_t(\gamma) = \underline{c} \;\Big|\; [\![x]\!]_t(\gamma) = \gamma(x) \;\Big|\; [\![t_1\ t_2]\!]_t(\gamma) = [\![t_1]\!]_t(\gamma)([\![t_2]\!]_t(\gamma))$$

$$\begin{array}{ll}
[\![\textbf{return}\ t]\!]_t(\gamma, dv) & = \delta_{[\![t]\!]_t(\gamma)}(dv) \\
[\![\textbf{do}\{x \leftarrow t; m\}]\!]_m(\gamma, dv) & = \int [\![t]\!]_t(\gamma, dz)[\![\textbf{do}\{m\}]\!]_t(\gamma[x \mapsto z], dv) \\
[\![\textbf{normal}\ t_1\ t_2]\!]_t(\gamma, dv) & = \mathcal{N}(v; [\![t_1]\!]_t(\gamma), [\![t_2]\!]_t(\gamma)) \cdot \Lambda(dv) \\
[\![\textbf{bernoulli}\ t]\!]_t(\gamma, dv) & = [\![t]\!]_t(\gamma) \cdot \delta_{\textbf{true}}(dv) + (1 - [\![t]\!]_t(\gamma)) \cdot \delta_{\textbf{false}}(dv)
\end{array}$$

Fig. 5. Semantics of the target language for the $d\{\cdot\}$ translation.

5 PROGRAM TRANSFORMATIONS FOR DENSITY ESTIMATION

In Section 4, we saw that λ_{SP} terms of type $D\,\sigma$, $P\,\tau$, and $M\,\tau$ can all be understood as denoting *measures* over particular spaces. Indeed, the semantics interprets these types using restricted spaces of well-behaved measures: measures that are absolutely continuous with respect to the reference measures ν_σ (for $D\,\sigma$) or $\nu_{\mathbb{T}}$ (for $P\,\tau$ and $M\,\tau$). This implies that all GenSP programs of these types denote measures that have *densities* with respect to their reference measures.

Computing these densities, however, may be intractable.[5] In this section, we present a program transformation $d\{\cdot\}$ (Fig. 6) that translates a λ_{SP} program into a new program that *estimates* the original program's density (or more specifically, that implements the stochastic probability interface described in Section 3, for the measure denoted by the original program).

Target language of the transformation. Our compiler translates terms from λ_{SP}'s rich source language into a standard higher-order functional language, with random sampling, but *without* λ_{SP}'s constructs for inference programming, **observe**, marginalization, or normalization. As such, the syntax of the target language inherits the *deterministic core* from Fig. 3, but features a much simpler probabilistic syntax and semantics (Fig. 5). The types $T\ \tau$ represent *target-language* probabilistic computations returning τ; semantically, T is the quasi-Borel subprobability monad.

The $d\{\cdot\}$ transformation on types. The top of Fig. 6 gives the action of $d\{\cdot\}$ on λ_{SP} types. A source-language term $\Gamma \vdash t : \tau$ is always translated into a target-language term $d\{\Gamma\} \vdash d\{t\} : d\{\tau\}$, where $d\{\Gamma\}$ is the target-language context obtained by applying $d\{\cdot\}$ to the types of all free variables in the source-language context Γ. In other words, $d\{\cdot\}$ translates terms of type τ into terms of type $d\{\tau\}$, assuming all the free variables also have translated types.[6]

Intuitively, $d\{\tau\}$ is the type of an *stochastic probability interface implementation* for a term of type τ. For ground types σ, $d\{\cdot\}$ is the identity: there is no special interface to implement for a non-probabilistic term. For products and functions, the $d\{\cdot\}$ translation is just pushed inward: $d\{\tau_1 \times \tau_2\} = d\{\tau_1\} \times d\{\tau_2\}$ and $d\{\tau_1 \to \tau_2\} = d\{\tau_1\} \to d\{\tau_2\}$. An interface implementation of a pair of programs is a pair of interface implementations; an interface implementation of a function is a function taking an implementation for its argument and returning an implementation for its result.

The first interesting type is $D\,\sigma$, for which we have $d\{D\,\sigma\} = (\sigma \to T\,\overline{\mathbb{R}}_{\geq 0}) \times T(\sigma \times \overline{\mathbb{R}}_{\geq 0})$. The first component of the translation is a positive unbiased density estimator for the distribution (Def. 3.1), and the second is a corresponding unbiased density sampler (Def. 3.2). The translations of $P\,\tau$ and $M\,\tau$ are slightly more involved. Our goal is to derive unbiased density estimators and samplers for the *trace* distributions they denote, but in order to do so compositionally, the program

[5] The intractability arises as a result of the **marginal** and **normalize** constructs. The density of **marginal** $p\ a$ is an integral over all traces of p, and the density of **normalize** $p\ a$ is an integral over the auxiliary variables sampled by the inference algorithm a. Without these constructs, exact densities are generally tractable; however, they are densities over high-dimensional trace spaces that many inference algorithms struggle to explore.

[6] This presentation is inspired by Huot et al. [2020]'s study of AD as a source-to-source "macro," much like our $d\{\cdot\}$.

Translating Types: if $\Gamma \vdash t : \tau$, *then* $d\{\Gamma\} \vdash d\{t\} : d\{\tau\}$

$$
\begin{array}{l|l}
d\{\sigma\} = \sigma & d\{D\,\sigma\} = (\sigma \to T\,\overline{\mathbb{R}}_{\geq 0}) \times T(\sigma \times \overline{\mathbb{R}}_{\geq 0}) \\
d\{\tau_1 \to \tau_2\} = d\{\tau_1\} \to d\{\tau_2\} & d\{P\,\tau\} = (\text{Trace} \to T\,(\overline{\mathbb{R}}_{\geq 0} \times \text{Trace})) \times T\,(\text{Trace} \times \overline{\mathbb{R}}_{\geq 0}) \times (\text{Trace} \to d\{\tau\}) \\
d\{\tau_1 \times \tau_2\} = d\{\tau_1\} \times d\{\tau_2\} & d\{M\,\tau\} = (\text{Trace} \to T\,(\overline{\mathbb{R}}_{\geq 0} \times \text{Trace})) \times (\text{Trace} \to d\{\tau\}) \\
 & d\{\text{Alg}\} = (\text{Trace} \to T\,\overline{\mathbb{R}}_{\geq 0}) \to d\{D\,\text{Trace}\}
\end{array}
$$

$T\ \sigma$ is a type representing *target-language* probabilistic computations whose densities need not be estimated.

Translating Terms

Core Calculus

$$
d\{c\} = c_d \;\Big|\; d\{x\} = x \;\Big|\; d\{t_1\ t_2\} = d\{t_1\}\ d\{t_2\} \;\Big|\; d\{\lambda x : \tau.t\} = \lambda x : d\{\tau\}.d\{t\} \;\Big|\; d\{(t_1, t_2)\} = (d\{t_1\}, d\{t_2\})
$$

$$
d\{\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2\} = \textbf{if } t \textbf{ then } d\{t_1\} \textbf{ else } d\{t_2\} \mid d\{\pi_i\ t\} = \pi_i d\{t\} \mid d\{\{t_1 \mapsto t_2\}\} = \{d\{t_1\} \mapsto d\{t_2\}\}
$$

Primitive Distributions (Selected Examples)

$$
\begin{aligned}
d\{\textbf{normal } t_1\ t_2\} &= (\lambda x.\textbf{return}(\mathcal{N}(x; t_1, t_2)), \textbf{do}\{x \leftarrow \textbf{normal } t_1\ t_2; \textbf{return}(x, \mathcal{N}(x; t_1, t_2))\}) \\
d\{\textbf{bernoulli } t\} &= (\lambda b.\textbf{return}(\textbf{if } b \textbf{ then } t \textbf{ else } 1 - t), \\
&\qquad \textbf{do}\{b \leftarrow \textbf{bernoulli } t; \textbf{return}(b, \textbf{if } b \textbf{ then } t \textbf{ else } 1 - t)\})
\end{aligned}
$$

Traced Programs

$$
\begin{aligned}
d\{\textbf{return } t\} &= (\lambda u.\textbf{return}(1, u), \textbf{return}(\{\}, 1), \lambda u.d\{t\}) \\
d\{\textbf{sample } t_d\ t\} &= (\lambda u.\textbf{do}\{w \leftarrow (\pi_1\ d\{t_d\})\ (u[t]); \textbf{return}(\mathit{has}_\sigma(u, t) \cdot w, u \setminus t)\}, \\
&\qquad \textbf{do}\{(x, w) \leftarrow \pi_2(d\{t_d\}); \textbf{return}(\{t \mapsto x\}, w)\}, \lambda u.u[t]) \\
d\{\textbf{observe } t_d\ t\} &= (\lambda u.\textbf{do}\{w \leftarrow (\pi_1\ d\{t_d\})\ t; \textbf{return}(w, u)\}, \lambda u.()) \\
d\{\textbf{do}\{x \leftarrow t; m\}\} &= (\lambda u.\textbf{do}\{(w, u') \leftarrow (\pi_1\ d\{t\})(u); \textbf{let } x = (\pi_3\ d\{t\})(u); \\
&\qquad\qquad (v, u'') \leftarrow (\pi_1\ d\{\textbf{do}\{m\}\})(u'); \textbf{return } (w \cdot v, u'')\}, \\
&\quad \textbf{do}\{(u_1, w_1) \leftarrow \pi_2 d\{t\}; \textbf{let } x = (\pi_3\ d\{t\})(u_1); \\
&\qquad (u_2, w_2) \leftarrow \pi_2 d\{\textbf{do}\{m\}\}; \textbf{if } \mathit{disj}(u_1, u_2) \textbf{ then return } (u_1 \mathbin{+\!\!+} u_2, w_1 \cdot w_2) \textbf{ else fail}\}, \\
&\quad \lambda u.\textbf{let } x = (\pi_3\ d\{t\})(u) \textbf{ in } (\pi_3\ d\{\textbf{do}\{m\}\})(u)) \\
d\{\textbf{do}\{t\}\} &= d\{t\}
\end{aligned}
$$

Inference Programming (Selected Examples)

$$
\begin{aligned}
d\{\textbf{marginal } t_p\ t\} &= (\lambda x.\textbf{let } p = \lambda u.\textbf{do}\{(w, \_) \leftarrow (\pi_1\ d\{M.\textbf{do}\{\mu \leftarrow t_p; \textbf{observe } \mu\ x\}\})(u); \textbf{return } w\} \textbf{ in} \\
&\qquad\quad \textbf{do}\{(u, w) \leftarrow \pi_2((d\{t\}\ x)\ p); v \leftarrow p(u); \textbf{return } (v \div w)\}, \\
&\quad \textbf{do}\{(u, w_1) \leftarrow \pi_2 d\{t_p\}; \textbf{let } \mu = (\pi_3 d\{t_p\})(u); (x, w_2) \leftarrow \pi_2(\mu); \\
&\qquad\quad \textbf{let } p = \lambda u.\textbf{do}\{(w, \_) \leftarrow (\pi_1\ d\{M.\textbf{do}\{\mu \leftarrow t_p; \textbf{observe } \mu\ x\}\})(u); \textbf{return } w\}; \\
&\qquad\quad v \leftarrow (\pi_1((d\{t\}\ x)\ p)(u); \textbf{return}(x, w_1 \cdot w_2 \div v)\}) \\
d\{\textbf{normalize } t_p\ t_i\} &= \textbf{let } a = d\{t_i\}(\lambda u.\textbf{do}\{(w, u') \leftarrow (\pi_1\ d\{t_p\})\ u; \textbf{return } (\mathit{isempty}(u') \cdot w)\}) \textbf{ in} \\
&\quad (\lambda u.\textbf{do}\{(\_, u') \leftarrow (\pi_1\ d\{t_p\})\ u; w \leftarrow \pi_1(a)(u \setminus u'); \textbf{return } (w, u')\}, \pi_2(a), \pi_2\ d\{t_p\}) \\
d\{\textbf{mcmc } t_p\ t_i\ t_n\} &= \lambda p.(\lambda u'.\textbf{do}\{w' \leftarrow p(u'); (u_0, w_0) \leftarrow \mathit{iterate}_T(\pi_2(d\{t_i\}(p)))\ t_n\ (u', w'); \\
&\qquad\qquad (q, u_q) \leftarrow (\pi_1\ d\{t_p\})(u_0); \textbf{return}(\mathit{isempty}(u_q) \cdot q \cdot w' \div w_0)\}, \\
&\qquad \textbf{do}\{(u_0, q) \leftarrow (\pi_2\ d\{t_p\}); w_0 \leftarrow p(u_0); \\
&\qquad\quad (u', w') \leftarrow \mathit{iterate}_T\ (\pi_1(d\{t_i\}(p)))\ t_n\ (u_0, w_0); \textbf{return}(q \cdot w' \div w_0)\}) \\
d\{\textbf{smc } t_i\} &= \lambda p.(\lambda u.\textbf{do}\{v \leftarrow p(u); (\mathbf{x}, j) \leftarrow (\pi_2\ d\{t_i\})\ (u, v); \\
&\qquad\qquad \mathbf{w}_{-j} \leftarrow \mathit{map}_T\ (\lambda(u_i, w_i, v_i).\textbf{do}\{v'_i \leftarrow p(u_i); \textbf{return}(w_i \cdot v'_i \div v_i)\})\ \mathbf{x}_{-j}; \\
&\qquad\qquad \textbf{return}(\mathit{mean}(\mathbf{w}_{-j} \mathbin{+\!\!+} [\pi_2\ \mathbf{x}_j]) \div v\}, \\
&\qquad \textbf{do}\{\mathbf{x} \leftarrow \pi_1\ d\{t_i\}; \\
&\qquad\quad \mathbf{w} \leftarrow \mathit{map}_T\ (\lambda(u_i, w_i, v_i).\textbf{do}\{v'_i \leftarrow p(u_i); \textbf{return}(w_i \cdot v'_i \div v_i, v'_i)\})\ \mathbf{x}; \\
&\qquad\quad \textbf{let } \hat{w} = \mathit{sum}(\mathit{map}\ \pi_1\ \mathbf{w}); i \leftarrow \textbf{categorical}(\mathit{map}\ (\lambda(w, v).w \div \hat{w})); \\
&\qquad\quad \textbf{return } (\pi_1(\mathit{nth}\ \mathbf{x}\ i), \hat{w} \div (\mathit{length}(\mathbf{w}) \cdot \pi_2(\mathit{nth}\ \mathbf{w}\ i)))\})
\end{aligned}
$$

Fig. 6. Stochastic Probability Interface compiler as a program transformation on λ_{SP}.

transformation must also track additional information. For $P\,\tau$, we have

$$d\{P\,\tau\} = (\text{Trace} \to T\ (\overline{\mathbb{R}}_{\geq 0} \times \text{Trace})) \times T\ (\text{Trace} \times \overline{\mathbb{R}}_{\geq 0}) \times (\text{Trace} \to d\{\tau\}).$$

The first component accepts as input a trace, and probabilistically outputs two values, a weight and another trace. If u is in the support of the distribution in question, the weight should be an unbiased density estimate of u. If $u = v +\!\!+\, v'$ for v in the support of the distribution, the second return value should be v', the trace of "left-over" values that the original program did not attempt to sample. The second component of the translation is the unbiased density sampler for the trace distribution, the one corresponding to the unbiased density estimator encoded by the first component. The third component is the (translation under $d\{\cdot\}$ of the) return-value function for the program. The case for $M\,\tau$ is similar, but because terms of type $M\,\tau$ denote unnormalized (and not probability) measures, we implement only unbiased density *estimators* for them, omitting the second component.

Translating primitives. We assume every primitive $c : \tau$ in λ_{SP} comes equipped with a valid implementation $c_d : d\{\tau\}$ of the SPI. For deterministic primitives, $c_d = c$, but for (e.g.) primitive distributions, like $c = \mathit{categorical}_n : \mathbb{R}^n \to D\,\mathbb{N}$, we must equip a non-trivial $c_d : \mathbb{R}^n \to ((\mathbb{N} \to T\overline{\mathbb{R}}_{\geq 0}) \times T\ (\mathbb{N} \times \overline{\mathbb{R}}_{\geq 0}))$, implementing some unbiased density estimator and sampler for the Categorical distribution. Our translation replaces c with its built-in SPI implementation c_d.

Translating sample and observe. For $t_d : D\,\sigma$ and t : Str, the program **sample** $t_d\ t$ is of type $P\,\sigma$. It encodes a distribution over singleton traces of the form $\{t \mapsto x\}$, where x is drawn from the distribution encoded by t_d; the density of a trace $\{t \mapsto x\}$ under the program is just the density of x under t_d. Because of this, the translation $d\{\textbf{sample}\ t_d\ t\}$, which is a tuple of three method implementations (see discussion of $d\{P\,\tau\}$ above), relies heavily on the translation of t_d, which implements the stochastic probability interface for t_d. The first method extracts the value at name t from the input trace u, uses $\pi_1 d\{t_d\}$ to estimate its density under t_d, and returns the density and the *left over* trace $u' = u \setminus t$ (the trace u with name t deleted). The second method uses t_d's unbiased density sampler, $\pi_2 d\{t_d\}$, to generate a pair (x, w), then wraps x in a trace $u = \{t \mapsto x\}$ before returning the pair (u, w). Finally, the third method implements the return-value function for the program, which given a trace $\{t \mapsto x\}$ returns x.

The program **observe** $t_d\ t$ (for $t_d : D\,\sigma$ and $t : \sigma$) is of type $M\,1$, and represents a *scaled* Dirac measure on the empty trace $\{\}$, with total mass equal to the density of t under t_d. The translation $d\{\textbf{observe}\ t_d\ t\}$ is a tuple of two method implementations (see discussion of $d\{M\,\tau\}$ above). The first method uses $\pi_1 d\{t_d\}$ to estimate the density of t under t_d, and returns the estimate and the input trace u, unchanged (since **observe** statements do not use any names in the trace). The second method is the return value function that maps any trace to () (**observe** statements are of unit type).

Translating do. If $t_p : P\,\tau_1$ and $x : \tau_1 \vdash \textbf{do}\{m\} : P\,\tau_2$, then $\textbf{do}\{x \leftarrow t_p; m\}$ has type $P\,\tau_2$ and represents a distribution over traces of the form $u_1 +\!\!+\, u_2$, where u_1 is a trace of the first part t_p of the computation, and u_2 is a trace of $\textbf{do}\{m\}$. The density of this distribution is the product of the densities of u_1 and u_2, and since the unbiased density estimates produced by $d\{t_p\}$ and $d\{\textbf{do}\{m\}\}$ are independent (conditioned on u_1 and u_2), their product unbiasedly estimates the density of the overall program. The translation of $\textbf{do}\{x \leftarrow t_p; m\}$ implements the required methods according to this logic, calling the corresponding methods for $d\{t_p\}$ and $d\{\textbf{do}\{m\}\}$ and multiplying the results.[7]

Translating inference algorithms. Terms of type Alg represent inference algorithms that can be applied to generate approximate posterior samples, given unnormalized target measures. When we translate a term of type Alg, our goal is to produce an implementation of the stochastic probability

[7] Fig. 6 shows how to translate **do** statements in the case where P is the monad in question; for M, the translation is the same, except that only the first and last components of the output tuple are generated.

interface *for the algorithm itself*, i.e., for the probability measure that the algorithm induces over approximate posterior traces. For **enumeration** (where applicable), this is the exact posterior; for **mcmc** $t_p\ t\ n$ it is the marginal distribution of the n^{th} state in the MCMC chain with transition kernel t, initialized with a sample from t_p; and for **smc** t it is the marginal distribution of a particle chosen from the final weighted particle collection generated by the SMC algorithm described by t. Formally, our translation produces a term of type $d\{\text{Alg}\} = (\text{Trace} \to T\,\overline{\mathbb{R}}_{\geq 0}) \to d\{D\,\text{Trace}\}$, where the input is the density estimator for a target measure, and the output is the stochastic probability interface implementation for the marginal distribution on output traces from the algorithm in question, applied to the given target.

These marginal distributions do have densities, but they can be very difficult to compute, arising as integrals over all of the random numbers generated during the course of the algorithm. Thankfully, techniques from the literature [Andrieu et al. 2010; Cusumano-Towner and Mansinghka 2017; Lew et al. 2022] can be adapted to derive estimators of these marginal densities. For example, to estimate the marginal density of an MCMC chain, we use the following result:

Proposition 5.1. *Let $K(x, dy)$ be a probability kernel that leaves a target measure π invariant, and let $L(y, dx)$ be its time reversal (so that $\pi(dy)L(y, dx) = \pi(dx)K(x, dy)$). Suppose $Q(dx)$ is a probability measure with $Q \ll \pi$ and $\pi \ll Q \ll \nu$ for a reference measure ν. Then $P(dx) = \int_{X^n} Q(dx_0) \prod_{i=1}^{n} K(x_{i-1}, dx_i)\delta_{x_n}(dx)$ is absolutely continuous with respect to ν, and the following yields an unbiased estimate of its density at a point x: let $x_n = x$, generate $x_{i-1} \sim L(x_i, dx_{i-1})$ for $i = 1, \ldots, n$, and then compute $\frac{dQ}{d\nu}(x_0)\frac{d\pi}{d\nu}(x_n)/\frac{d\pi}{d\nu}(x_0)$.*

When translating **mcmc** $t_p\ t\ n$, the term t_p plays the role of the initial distribution Q, and its translation $d\{t_p\}$ lets us estimate its density unbiasedly. The term t, of type *MCMC* (representing MCMC kernels) plays the role of K, and *its* translation lets us run the time-reversal kernel L.[8] The translation of **smc** is based on a similar result, allowing the density of a point x to be estimated by combining an estimate of $\pi(x)$ (the target density) with the average weight computed by the *conditional* version of the SMC algorithm with retained particle x [Andrieu et al. 2010].[9]

Translating marginalization and normalization. For $t_p : P\,(D\,\sigma)$, the term **marginal** $t_p\ a$ represents the marginal distribution that arises by generating a trace of t_p, computing the return value $d : D\,\sigma$ of the program on that trace, and sampling a value $x \sim d$. The density of this distribution is the *integral* of the density of d, over all traces of t_p. Our translation estimates this integral with importance sampling, with the inference algorithm a as the proposal distribution.

More precisely, suppose we wish to estimate the density of a point $x \in [\![\sigma]\!]_m$. Our algorithm first defines $p : \text{Trace} \to T\,\overline{\mathbb{R}}_{\geq 0}$, a density estimator for a target measure (over traces) whose normalizing constant is precisely the density we wish to estimate. To define p, our algorithm translates the term $M.\mathbf{do}\{d \leftarrow t_p; \mathbf{observe}\ d\ x\}$ and uses the resulting density estimator on the input trace u. We then instantiate the (translated) user's inference algorithm $d\{a\}$ on the value x and the density estimator p, to obtain an approximate posterior for the target. We use this approximate posterior as a proposal, calling its unbiased density sampler to generate a pair (u, w) of a proposed trace u and an estimate w of its density, then running $p(u)$ to obtain the numerator of the importance weight. Because this density estimator is based on importance sampling, we are able to automate the corresponding unbiased density sampler according to the logic in Example 3.2.

The term **normalize** $t_p\ a$ denotes the approximate posterior of t_p under the inference algorithm a. But the density of this measure is already estimated by the translation of a, applied to the target

[8] This is because $d\{MCMC\}$ is a product type giving implementations of both the kernel K and its time reversal L. The time reversal of an MH kernel is itself; the time reversal of **sequence** $t_1\ t_2$ is the reversed sequence.

[9] Just as $d\{MCMC\}$ compositionally constructs time reversals, $d\{SMC\}$ constructs conditional SMC algorithms.

Logical Relations $R_\tau \subseteq [\![\tau]\!]_m \times [\![\tau]\!]_d$

$$\begin{aligned}
(x, y) \in R_\sigma &\iff x = y \\
((x_1, x_2), (y_1, y_2)) \in R_{\tau_1 \times \tau_2} &\iff (x_1, y_1) \in R_{\tau_1} \wedge (x_2, y_2) \in R_{\tau_2} \\
(f, g) \in R_{\tau_1 \to \tau_2} &\iff \forall (x, y) \in R_{\tau_1}.(f(x, y), g(y)) \in R_{\tau_2} \\
(\mu, (\xi, \chi)) \in R_{D\,\sigma} &\iff AC_{D\,\sigma}(\xi, \chi) \implies SPI_{D\,\sigma}(\mu, (\xi, \chi)) \\
((\mu, f), (\xi, \chi, g)) \in R_{P\,\tau} &\iff \forall u \in \mathbb{T}.(f(u), g(u)) \in R_\tau \wedge DS(\mu) \wedge (AC_P(\xi, \chi) \implies SPI_P(\mu, (\xi, \chi))) \\
((\mu, f), (\xi, g)) \in R_{M\,\tau} &\iff \forall u \in \mathbb{T}.(f(u), g(u)) \in R_\tau \wedge DS(\mu) \wedge (AC_M(\xi) \implies SPI_M(\mu, \xi)) \\
(\alpha, \beta) \in R_{\mathrm{Alg}} &\iff \forall \xi \in \mathbb{T} \Rightarrow \mathrm{Prob}\ \overline{\mathbb{R}}_{\geq 0}.\forall \mu.UB_{\mathrm{Trace}}(\mu, \xi) \implies SPI_{\mathrm{Alg}}(\alpha(\xi), \beta(\xi), \mu) \\
(\mathcal{A}, (\mathcal{A}', \mathcal{A}^\dagger, \xi)) \in R_{SMC} &\iff \mathcal{A} = \mathcal{A}' \wedge AC_{SMC}(\mathcal{A}, \mathcal{A}^\dagger) \implies CSMC(\mathcal{A}, \mathcal{A}^\dagger, \xi) \\
(k, k') \in R_{MCMC} &\iff k = \pi_1 \circ k' \wedge \forall (\mu, \xi).(UB_{\mathrm{Trace}}(\mu, \xi) \wedge AC_{MCMC}(k'(\xi), \xi)) \implies TR(k'(\xi), \xi)
\end{aligned}$$

Auxiliary Definitions

The notation "$\dot\forall_\mu x \in X$" means "for μ-almost-all $x \in X$", where if $X = [\![\sigma]\!]_m$ and μ is omitted, it is understood to be ν_σ. $|\mu|$ is the total mass of μ; $\mathbb{T}_\mu = \{u \in \mathbb{T} \mid \exists u'.\frac{d\mu}{d\nu_{Trace}}(u +\!\!+\, u') > 0\}$; and $\mathbb{T}_\mu(u)$ is the largest subtrace of u in $\mathbb{T}_\mu$.

$$\begin{aligned}
AC_{D\,\sigma}(\xi, \chi) &\iff \dot\forall x \in [\![\sigma]\!]_m.\ |\xi(x)| = |\chi| = 1 \\
AC_M(\xi) &\iff \dot\forall x \in \mathbb{T}.\ |\xi(x)| = 1 \\
AC_P(\xi, \chi) &\iff \dot\forall x \in \mathbb{T}.\ |\xi(x)| = |\chi| = 1 \\
AC_{SMC}(\mathcal{A}, \mathcal{A}^\dagger) &\iff \dot\forall_{\mathcal{A}}(x_i, w_i, v_i)_{i \in I} \in \mathrm{List}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}).\forall i \in I.w_i \neq 0 \implies |\mathcal{A}^\dagger((x_i, v_i))| = |\mathcal{A}| = 1 \\
AC_{MCMC}((k, l), \xi) &\iff \dot\forall_{\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi)}(x, w) \in \mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}.|k(x, w)| = |l(x, w)| = 1 \\
DS(\mu) &\iff \dot\forall_{\mu \otimes \mu}(t_1, t_2) \in \mathbb{T} \times \mathbb{T}.t_1 \neq t_2 \implies \exists s \in [\![\mathrm{Str}]\!]_m.s \in t_1 \wedge s \in t_2 \wedge t_1[s] \neq t_2[s] \\
UB_\sigma(\mu, \xi) &\iff \dot\forall x \in [\![\sigma]\!]_m.\mathbb{E}_{w \sim \xi(x)}[w] = \frac{d\mu}{d\nu_\sigma}(x) \wedge \dot\forall_\mu x \in [\![\sigma]\!]_m.\mathbb{P}_{w \sim \xi(x)}[w > 0] = 1 \\
SPI_{D\,\sigma}(\mu, (\xi, \chi)) &\iff UB_\sigma(\mu, \xi) \wedge \chi = (\lambda(x, w).w) \odot (\nu_\sigma \otimes \xi) \\
SPI_P(\mu, (\xi, \chi)) &\iff SPI_{D\ \mathrm{Trace}}(\mu, (\pi_{1*}\xi|_{\mathbb{T}_\mu}, \chi)) \wedge \dot\forall_\mu u \in \mathbb{T}.\forall u' \in \mathbb{T}.disj(u, u') \implies \pi_{2*}\xi(u +\!\!+\, u') = \delta_{u'} \\
&\qquad \wedge \forall u \notin \mathbb{T}_\mu, \pi_{1*}\xi(u) = \pi_{1*}\xi(\mathbb{T}_\mu(u)) \\
SPI_M(\mu, \xi) &\iff UB_{\mathrm{Trace}}(\mu, \pi_{1*}\xi|_{\mathbb{T}_\mu}) \wedge \dot\forall_\mu u \in \mathbb{T}.\forall u' \in \mathbb{T}.disj(u, u') \implies \pi_{2*}\xi(u +\!\!+\, u') = \delta_{u'} \\
&\qquad \wedge \forall u \notin \mathbb{T}_\mu, \pi_{1*}\xi(u) = \pi_{1*}\xi(\mathbb{T}_\mu(u)) \\
SPI_{\mathrm{Alg}}(\alpha, \beta, \mu) &\iff AC_{D_{\mathrm{Trace}}}(\beta) \implies (\mu \ll \alpha \wedge \alpha \ll \mu \wedge SPI_{D\ \mathrm{Trace}}(\alpha, \beta)) \\
CSMC(\mathcal{A}, \mathcal{A}^\dagger, \xi) &\iff \exists \mu.UB_{\mathrm{Trace}}(\mu, \xi) \wedge \dot\forall_{\mu \otimes \xi}(u, v) \in \mathbb{T} \times \overline{\mathbb{R}}_{\geq 0}.\mathbb{P}_{(\mathbf{s}, j) \sim \mathcal{A}^\dagger(u, v)}[\pi_{1,2}(\mathbf{s}_j) = (u, v)] = 1 \\
&\qquad \wedge \dot\forall_{\mathcal{A}}\mathbf{s} \in \mathrm{List}\ (\mathbb{T} \times \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}).\forall j \leq |\mathbf{s}|.\frac{d((\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi))\mathcal{A}^\dagger)}{d(\mathcal{A} \otimes \Phi)}(\mathbf{s}, j) = \tfrac{1}{|\mathbf{s}|}\textstyle\sum_{i=1}^{|\mathbf{s}|} \pi_3(\mathbf{s}_i) \\
TR((k, l), \xi) &\iff (\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi)) = k_*(\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi)) \\
&\qquad \wedge (\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi)) \otimes k = swap_*((\pi_2 \odot (\nu_{\mathrm{Trace}} \otimes \xi)) \otimes l)
\end{aligned}$$

Fig. 7. Our logical relations (top) can be read as specifications for the $d\{\cdot\}$ translation: a term t of type τ is correctly translated if $([\![t]\!]_m(\gamma_m, \gamma_d), [\![d\{t\}]\!]_t(\gamma_d)) \in R_\tau$ when $(\gamma_m, \gamma_d) \in R_\Gamma$. In these specs, we employ several auxiliary definitions, for absolute continuity (AC), discrete structure (DS), unbiasedness (UB), satisfaction of the stochastic probability interface (SPI), conditional sequential Monte Carlo (CSMC), and time-reversal (TR).

density estimator given by translating t_p. Our translation of **normalize** t_p a, then, applies $d\{a\}$ to $d\{t_p\}$ (and performs some projections and bookkeeping to make the types work out, since **normalize** t_p a has type $P\,\tau$, whereas $d\{a\}$ yields an estimator for a $d\{D\,\mathrm{Trace}\}$).

Correctness. Our translation is correct—in the sense that it implements correct unbiased density estimators for the measures denoted by user programs—under an *absolute continuity* condition on the user's program. Informally, the condition states that every time the user chooses a custom proposal distribution for an inference algorithm, the custom proposal has the right support. We note that users of Gen, Pyro, and similar languages must also hand-verify this condition to ensure that their inference algorithms are correct; it is not a *new* requirement of our translation. In fact, researchers have developed a number of techniques for statically checking the requirement in restricted languages [Lee et al. 2020; Lew et al. 2020; Li et al. 2023; Wang et al. 2021].

To encode this requirement, we can augment our translation with *absolute continuity assertions* $\textbf{assert}_{\ll}\ p\ q : T\,1$, which probabilistically test absolute continuity of p with respect to q. The assertions are not just theoretical: they can be implemented, and serve as probabilistic *tests* that the absolute continuity condition holds, passing with probability 1 if and only if it does (Appx. D). In our semantics, the assertion denotes a subprobability measure that assigns to the unit value () mass equal to the probability of the test's passing. We can then define what it means for a term t to satisfy the absolute continuity condition, based on the fact that if the assertion ever fails, the failure will "shave off probability mass" from $d\{t\}$, and as a result, the translation will denote a subprobability kernel or measure, rather than a probability kernel or measure.

Definition 5.1. *We say a closed term t satisfies the absolute continuity condition if it meets the following criterion depending on its type:*

- *For $t : D\,\sigma$, $\pi_1[\![d\{t\}]\!]_t$ must be a probability kernel, and $\pi_2[\![d\{t\}]\!]_t$ must be a probability measure.*
- *For $t : P\,\tau$ or $\vdash t : M\,\tau$, $\pi_1[\![d\{t\}]\!]_t$ must be a probability kernel.*
- *For $t : P\,\tau$, $\pi_2[\![d\{t\}]\!]_t$ must additionally be a probability measure.*

Using this definition, we can state the following theorem:

Theorem 5.2. *The translation $d\{\cdot\}$ is sound, in the following sense:*

(1) If $\vdash t : D\,\sigma$ is a closed term satisfying the absolute continuity condition, then $[\![d\{t\}]\!]_t$ is an implementation of the full stochastic probability interface for $[\![t]\!]_m$ (cf. Sec. 3).

(2) If $\vdash t : P\,\tau$ or $\vdash t : M\,\tau$, and t satisfies the absolute continuity condition, then

$$[\![\lambda u.\textbf{do}\{(w,u') \leftarrow (\pi_1 d\{t\})\ u;\ \textbf{if}\ \mathit{isempty}\ u'\ \textbf{then return}\ w\ \ \textbf{else return}\ 0\}]\!]_t$$

is an unbiased density estimator for $\pi_1[\![t]\!]_m$.

(3) If $\vdash t : P\,\tau$ satisfies the absolute continuity condition, then $\pi_2[\![d\{t\}]\!]_t$ is the corresponding unbiased density sampler for the density estimator in (2).

Our proof (Appx. E) uses logical relations: in Fig. 7, we define relations $R_\tau \subseteq [\![\tau]\!]_m \times [\![\tau]\!]_d$ that encode correctness specifications for $d\{\cdot\}$ at types τ. We prove by induction that each term constructor in our language preserves correctness of the translation: on pairs of environments $\overline{\gamma}$ where each entry $x_i : \tau_i$ is such that $(\overline{\gamma}_m(x_i), \overline{\gamma}_d(x_i)) \in R_{\tau_i}$, we have $([\![\Gamma \vdash t : \tau]\!]_m(\overline{\gamma}), [\![\Gamma \vdash t : \tau]\!]_d(\overline{\gamma}_d)) \in R_\tau$. The final result follows by instantiating correctness preservation for closed terms of type $D\,\sigma$, $P\,\tau$, and $M\,\tau$.

6 EVALUATION

We evaluate the performance of our approach using GenSP, a version of the Gen.jl probabilistic programming library for Julia [Cusumano-Towner 2020], extended to support our novel constructs.

Benchmarks. Our evaluation is based on a selection of inference problems from the literature; for each, we implement a model and inference algorithm in GenSP.

- **3DP3**: We implement Gothoskar et al. [2021]'s model of multi-object 3D scenes, based on a prior over *scene graphs* [Johnson et al. 2018, 2015], whose nodes are 3D objects from the YCB database [Calli et al. 2015], and whose edges encode contact relationships and relative poses. The inference task is to infer the scene graph from a point cloud captured by a depth camera. We use the same custom MCMC proposals as in Gothoskar et al. [2021], but we use a GenSP density estimator for the likelihood, which is implemented as the **marginal** of a process that repeatedly generates noisy points from a latent cloud. As a result, the algorithm becomes a *pseudomarginal* MCMC algorithm [Andrieu and Roberts 2009; Beaumont 2003; Doucet et al. 2015].

Table 1. Microbenchmarks for GenSP density estimators. Runtime of each density implementation for seven distributions from our case studies. For each distribution, runtime is reported for several typical density queries, with inputs of varying size. The mark ✗ indicates that no exact density evaluator is available.

benchmark	runtime Exact[1]	 Handcoded[2]	 GenSP.jl[3]	speedup vs. exact	overhead vs. handcoded
3DP3 [Gothoskar et al. 2021]					
5 objects, 2.1k points, 1.7k observed	984 ± 9 ms	57 ± 7 ms	40 ± 6 ms	**17x**	1.4x
2D cloud, 18.2k points, 34.9k observed	47 ± 2 s	0.87 ± 0.05 s	1.04 ± 0.02 s	**45x**	1.2x
Context-Correct [Mays et al. 1991]					
6 letters, edit distance 2	✗	14 ± 3 μs	47 ± 204 μs	∞	3.6x
6 letters, edit distance 5	✗	90 ± 477 μs	220 ± 484 μs	∞	2.4x
10 letters, edit distance 2	✗	19 ± 6 μs	65 ± 500 μs	∞	3.4x
Context-Correct (trunc.) [Mays et al. 1991]					
6 letters, edit distance 2, max typos 2	1.06 ± 0.003 s	13 ± 1 μs	45 ± 153 μs	**23,555x**	3.4x
6 letters, edit distance 5, max typos 2	1.10 ± 0.01 s	52 ± 379 μs	130 ± 803 μs	**2,068x**	2.5x
10 letters, edit distance 2, max typos 2	3.89 ± 0.13 s	19 ± 6 μs	59 ± 212 μs	**204,736x**	3.1x
4 letters, edit distance 3, max typos 3	145 ± 1 s	9 ± 102 μs	40 ± 335 μs	**3,625,000x**	4.4x
RAVI-DPMM [Lew et al. 2022]					
10 datapoints, 7 merges (good clustering)	2.51 ± 0.01 s	2.7 ± 1.6 ms	4.9 ± 2.3 ms	**512x**	1.8x
40 datapoints, 37 merges (good clustering)	> 10 min	168 ± 4 ms	243 ± 6 ms	**>2,500x**	1.4x
40 datapoints, 39 merges (bad clustering)	> 10 min	235 ± 7 ms	297 ± 7 ms	**>2,020x**	1.3x
RSA [Goodman and Frank 2016]					
depth 1	13 ± 4 ms	131 ± 422 μs	171 ± 480 μs	**76x**	1.3x
depth 2	11 ± 0.1 s	1.3 ± 1.4 ms	1.7 ± 1.5 ms	**6,470x**	1.3x
RANSAC-Regress [Fischler and Bolles 1981]					
10 points, subset of 3	0.94 ± 1.3 ms	6 ± 24 μs	29 ± 182 μs	**32x**	4.8x
10 points, subset of 5	1.7 ± 1.3 ms	9 ± 52 μs	28 ± 137 μs	**60x**	3.1x
100 points, subset of 3	805 ± 233 ms	8 ± 43 μs	28 ± 153 μs	**28,750x**	3.5x
100 points, subset of 5	> 10 min	7 ± 29 μs	31 ± 207 μs	**>19,000,000x**	4.4x
Goal-Infer [Cusumano-Towner et al. 2017]					
300 RRT iters, 3500 refinements, likely dest.	✗	959 ± 252 μs	5.2 ± 1.5 ms	∞	5.4x
300 RRT iters, 3500 refinements, unlikely dest.	✗	334 ± 237 μs	4.0 ± 1.5 ms	∞	11.9x

[1]: exact density evaluators hand-coded in Julia, against the Gen.jl distribution interface, to expose as Gen primitives
[2]: density estimators hand-coded in Julia, against the GenSP.jl distribution interface, to expose as GenSP primitives
[3]: density estimators implemented within GenSP's unbiased-by-construction estimator DSL, exploiting automation

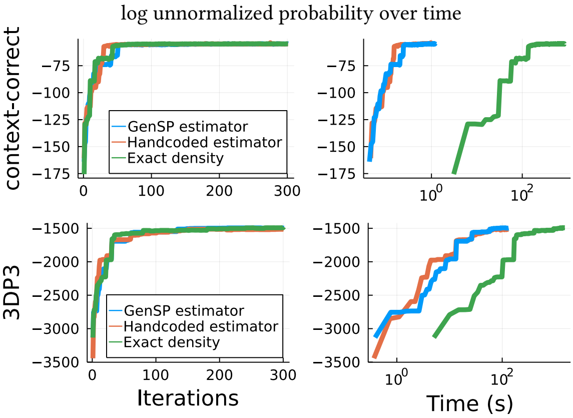
- **Context-Correct**: A context-sensitive spelling correction task [Mays et al. 1991], applied to retyped sentences from telenovela screenplays. In our model of typos, each word is corrupted by a sequence of zero or more insertions, deletions, and substitutions; **marginal** is used to marginalize out the sequence of edits. We implement both MCMC inference and SMC inference that fixes one word at a time. Due to the estimated likelihood, these are instances of pseudomarginal MCMC [Andrieu and Roberts 2009] and *random-weight* particle filtering [Fearnhead et al. 2010].
- **RSA**: We implement a model of pragmatic language understanding, based on the Rational Speech Acts framework [Goodman and Frank 2016]. Unlike in typical formulations of RSA, in our GenSP model, agents reason about one another using *approximate* inference, encoded via **normalize**. As such it can be seen as a boundedly rational version of the model [Zhi-Xuan et al. 2020]. GenSP automatically estimates the densities of the approximate inference routines used by the agents. The resulting inference algorithm is an instance of IS^2 [Tran et al. 2013], because it nests an importance sampling estimator of the likelihood within importance sampling.
- **RANSAC-Regress**: A standard Bayesian regression model, but with inference based on the RANSAC algorithm [Cusumano-Towner and Mansinghka 2018; Fischler and Bolles 1981]. RANSAC chooses a small subset of the data, fits model parameters to it, and proposes nearby parameters from a Gaussian; we use **marginal** to marginalize out the choice of data subset. As a result, the proposal density is estimated by GenSP, and the overall algorithm is an instance of random-weight [Chopin et al. 2020, Chapter 8] or RAVI [Lew et al. 2022] importance sampling.
- **Goal-Infer**: A model of an agent planning a path in a room, where the goal of inference is to infer the agent's destination from observations of their movements [Cusumano-Towner 2020, Chapter 6]. The model is a GenSP program that uses **marginal** to marginalize out the agent's

path-planning choices, which the agent makes using a randomized path planner [Zucker et al. 2007]. The resulting algorithm is an instance of IS^2 [Tran et al. 2013].
- **RAVI-DPMM**: A Bayesian agglomerative clustering algorithm from Lew et al. [2022], applied to astronomy data. The model is a standard Dirichlet process mixture model [Neal 2000], but importance sampling inference is performed using an involved proposal that runs a randomized agglomerative clustering algorithm on the data to propose a partition. As in Lew et al. [2022], we use a GenSP density estimator for the *proposal*, whose exact marginal density is an intractable sum over all possible sequences of merges that the agglomerative clustering algorithm makes. Our GenSP estimator uses **smc** within **marginal** to estimate the proposal's marginal density.

Experiments. Our evaluation is designed to answer two questions:

- *How do our sound-by-construction inference algorithms, with automated density estimators, perform, compared to hand-coded versions of the same algorithms?* Without GenSP, if practitioners wish to use fast density estimators within their inference algorithm implementation, they must hand-code the density estimators and ensure that they are unbiased. This can be time-consuming and error-prone, especially since there is no easy way to unit-test the unbiasedness of hand-coded estimators. GenSP provides a language for exploring the space of unbiased density estimation strategies, and convenient, correct automation. However, it also introduces overhead. **Our first experiment measures the overhead of GenSP's density estimators compared to hand-coded versions.** We used Julia to implement hand-coded versions of each density estimator automated by GenSP. We manually derived expressions for the density estimates, and wrote code to compute them, performing several manual optimizations (e.g., avoiding the computation of a density factor if it appeared both in the numerator and denominator of a density ratio, and using local variables to store random samples instead of heap-allocated traces). We measure both the overhead of each density query, and the overhead of the entire inference algorithm.
- *How does the noise introduced by stochastic probability estimators impact the accuracy of inference, compared to an idealized algorithm using exact inference?* GenSP's fast unbiased density estimation makes it possible to run inference using models and proposals for which it would be impossible, or impossibly slow, to evaluate exact densities. Thm. C.1 proves inference is still sound, i.e., accurate in the limit of infinite MCMC iterations or SMC particles. But a natural question is whether inference accuracy with finite computation suffers as a result of using these stochastic estimators. **Our second experiment measures the convergence rates of MCMC and SMC using exact vs. stochastic densities, in terms of *number of iterations or particles*.** When exact densities are available but slow, we use them; when they are unavailable, we approximate them using estimates with very high particle counts, which we chose to ensure negligible variance. We also report wall-clock times for the GenSP and exact variants of each density and inference algorithm; these help illustrate that even when GenSP's convergence may be slightly slower as a function of iteration or particle count, it is significantly faster in wall-clock time.

Results: Overhead. Table 1 shows microbenchmarks from our first experiment. The overhead of our automation is generally 1-5x for individual density queries, but *end-to-end* overhead is lower: in Figs. 8 and 9, the righthand plots show that in wall-clock time, the GenSP inference algorithms are only 1-2x slower than the hand-coded versions. This is because our fast unbiased estimators account for relatively little of the inference runtime in these benchmarks. For example, in the Context-Correct benchmark, one iteration of MCMC takes about 3ms, of which 500μs (16%) is density evaluation. Using a hand-coded density cuts this to about 150μs, but an iteration still takes about 2.65ms; thus, inference with GenSP is only 1.13x slower than hand-coded. Profiling suggests future engineering could mitigate the sources of overhead, which include the construction of trace data structures, and the failure to automatically 'cancel' (and thus avoid computing) terms



	Exact	GenSP
Context-Correct	41.1%	37.6%
3DP3	25.0%	18.5%

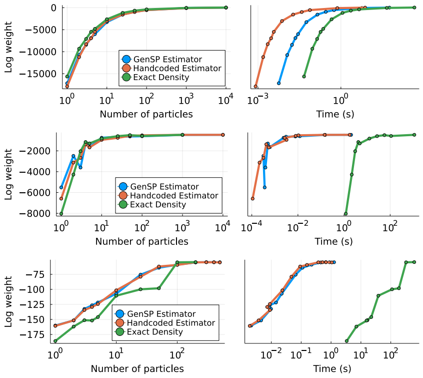
MH acceptance rates. Each iteration of Context-Correct performs 6 Metropolis-Hastings moves, and each iteration of 3DP3 performs 45. We report the percentage of iterations on which at least one proposal was accepted, across 10 runs of 300 iterations each. On average, using estimated densities led to a 10-25% reduction in acceptance rate, due to noise in the computation of the acceptance probability. Margin of error is about ±3% in all four measurements.

Fig. 8. Impact of GenSP density estimators on convergence of MCMC inference. *Left:* We plot exact log probability vs. # of iterations, for MCMC algorithms for context-sensitive spelling correction and 3D scene understanding. *Right:* For each algorithm, we report the average acceptance probability across its run.

Goal-Infer (expensive, low-variance estimator used in lieu of intractable exact density)

RANSAC-LinReg

Context-Correct (truncated exact density)



	Exact	GenSP
Context-Correct	16.8	22.1
RAVI-DPMM	6.3	6.4
Goal-Infer	49.8	64.9
RANSAC-LinReg	1521	2402

Relative variance of particle weights. The number of particles required for accurate inference depends on the variance of the importance weights. Two factors contribute to variance [Lew et al. 2022]: the quality of the proposal distribution (equal across settings), and the noise in the computation of densities (higher for GenSP).

Fig. 9. Impact of GenSP density estimators on convergence of SMC and IS inference. *Left:* We plot average log marginal likelihood estimate vs. # of particles, for SMC algorithms for context-sensitive spelling correction, goal inference, and robust regression. *Right:* We report the estimated variance of particle weights when exact densities vs. GenSP density estimators are used. *Both:* When exact densities were unavailable or timed out, we instead used estimated densities with enough replicates to eliminate non-negligible variance.

that match in the numerators and denominators of density ratios. For example, one optimization might fuse the bodies of proposal and model programs, eliminating common sub-expressions, and removing the need to construct traces to communicate between the two programs; local variables could be used directly to remember the proposed variables and score them under the model.

Results: Speed and convergence. The use of GenSP's unbiased density estimates does reduce the acceptance probability (in Metropolis-Hastings) and increases the variance of particle weights (in SMC), but these effects appear small and do not significantly affect the number of iterations or particles necessary for convergence, as shown in the plots in Figures 8 and 9. In the righthand plots, it can be seen that in wall-clock time the GenSP algorithms converge significantly faster than their exact-density counterparts. This is also reflected in Table 1, which shows orders-of-magnitude speedups over exact densities when they are available. We also see that our unbiased estimators scale favorably with problem size, whereas exact densities scale poorly.

7 RELATED WORK

Programmable inference. We build on a long line of work in probabilistic programming that aims to make inference *programmable* [Bingham et al. 2019; Cusumano-Towner 2020; Cusumano-Towner et al. 2019; Ge et al. 2018; Lew et al. 2021; Mansinghka et al. 2014, 2018; Narayanan and Shan 2020; Shan and Ramsey 2017; Stites et al. 2021; Wang et al. 2021; Zinkov and Shan 2016]. Our core calculus λ_{SP} is closest to languages like Gen [Cusumano-Towner et al. 2019], Pyro [Bingham et al. 2019], and ProbTorch [Stites et al. 2021]. Unlike these languages, we support models and proposals with estimated densities, letting us expose new **normalize** and **marginal** operations. As we discuss in Sec. 8, we believe that each of these systems could be extended to support our Stochastic Probability Interface, and versions of our **marginal** and **normalize** features; indeed, we have already extended Gen in this way, in GenSP. Many existing systems for programmable inference handle variational inference, whereas this paper has focused on Monte Carlo. To support variational inference, we could apply ADEV [Lew et al. 2023] to the stochastic density estimators we automate, to yield unbiased gradient estimates of lower bounds on the ELBO [Lew et al. 2022, Theorem 4].

We also take inspiration from inference combinators [Stites et al. 2021], and compositional inference frameworks like `monad-bayes` [Ścibior et al. 2018], which explicitly build sound samplers by transforming existing samplers, via combinators or monad transformers. Indeed, the λ_{SP} constructs for building terms of type `SMC` are very similar to Stites et al. [2021]'s combinators for soundly composing properly weighted population samplers. But unlike in that work, our transformation $d\{\cdot\}$ operates on terms of type `SMC` to automatically derive corresponding *conditional* SMC algorithms, which enables us to unbiasedly estimate the density of the user's SMC algorithm itself.

Some instances of algorithms that we have highlighted, e.g. pseudomarginal MH, can also be expressed compositionally in other languages. For example, `monad-bayes` [Ścibior et al. 2018] can implement particle-marginal MH by composing SMC and MCMC monad transformers. However, `monad-bayes` does not permit customization of the SMC proposals or the MH proposals used in the algorithm, and so could not express many of our case studies. As another example, by introducing auxiliary variables into models or proposals in the inference combinators framework, users can implement certain random-weight particle filters [Fearnhead et al. 2010]. However, that work supports only one strategy for approximately marginalizing the auxiliary variables, which often leads to high-variance weights. By contrast, our **marginal** construct allows users to specify any inference algorithm for marginalization, enabling significant variance reduction.

Automating density computations. The problem of automating *exact* marginal density evaluation for probabilistic programs is well-studied. The Hakaru system [Narayanan and Shan 2020; Shan and Ramsey 2017] is an interesting example, because it outputs densities as *probabilistic programs* of type $M\,1$, and such terms can be read *either* as exact integral expressions *or* as unbiased density estimators. Hakaru does not produce unbiased density *samplers* and thus cannot be used to, e.g., estimate proposal densities within importance sampling or Metropolis-Hastings. Hakaru also does not provide users with levers for controlling the variance of the resulting density estimators, which can be high. It would be interesting to investigate further whether the Hakaru approach can be extended with these features. Hakaru already has some features that would be intriguing to bring to GenSP, including support for automating certain change-of-variable corrections.

Nested inference. First-class inference within models was a key feature of Church [Goodman et al. 2008], which used rejection sampling to implement exact inference at both the outer and inner levels. Researchers have since proposed dynamic programming for faster nested inference [Stuhlmüller and Goodman 2012], approximate inference methods for certain forms of nested inference [Rainforth 2018], and semantic analyses [Zhang and Amin 2022]. We also study **normalize** as a first-class construct, but for us, it denotes *approximate* inference, and we use it in both models and proposals.

8 CONCLUSION

This paper proposes a key change to standard PPL architectures: replace *exact density evaluators* with the *stochastic probability interface* (Sec. 3). Although our presentation is compiler-based (Sec. 5), the SPI is also compatible with PPLs based on effect handlers, monad transformers, or macros.

For example, consider Python PPLs such as Pyro [Bingham et al. 2019] or ProbTorch [Stites et al. 2021], which typically feature *distribution* classes with methods for simulation and exact density evaluation. A first step toward adopting the SPI is to replace these methods with those we introduce in Sec. 3 for estimating densities and their reciprocals. Then, using non-standard execution of user probabilistic code (e.g., Pyro's "Poutine" library), existing density accumulation passes could be changed to accumulate density *estimates*, following the logic of Sec. 5's compiler on the constructs **sample**, **observe**, **return**, and **do**. Appx. C then gives the necessary changes to inference algorithms to make them work soundly with these stochastic density estimators.

Constructs for *marginalization* and *approximate normalization* can then be added as new classes implementing the SPI, with constructors that take as input the program to be marginalized or normalized. These classes could use any sound strategy for estimating the necessary densities. Our approach is to give the user a *choice* of inference method at each **marginal** and **normalize**.

Indeed, by selecting different estimation strategies at different points in the program, users can explore a vast array of unbiased-by-construction density estimation strategies, each striking different runtime-variance tradeoffs. We are interested to explore this design space; the fact that every application of **marginal** and **normalize** introduces approximation (and therefore variance) means that the usual advice about the value of these operations may not always apply in our setting. For example, the Rao-Blackwell theorem often implies that marginalizing variables from a model should reduce variance of the overall sampler; but to what extent does that logic apply if the marginal densities in question are only approximate? It would be interesting to extend recent results characterizing the variance of pseudomarginal importance samplers [Lew et al. 2022] to the more general setting we explore here. In any case, there is ample evidence from the machine learning and statistics literatures that various points along the spectrum—from exact but expensive marginal densities, to cheap but higher-variance estimates—are worth exploring. In a recent book on Monte Carlo inference, Chopin et al. [2020] called the technique of stochastically estimating densities within inference "arguably one of the most productive ideas in stochastic simulation, [resulting] in a multitude of practically useful algorithms." We hope this research will bring welcome PPL automation to practitioners who rely on these techniques—and conversely, open up this broad class of practically useful models and inference algorithms to probabilistic programmers.

Beyond expressiveness, our approach may also open up new opportunities for fine-grained parallelism in inference, because each call to **marginal** or **normalize** defines an inference subproblem. If we can build optimizing compilers that generate fast unbiased density estimators for modern parallel hardware, we are excited to see what new inference problems might then be in reach.

ACKNOWLEDGMENTS

We are grateful to Ben Sherman, Michael Carbin, Marco Cusumano-Towner, Alexey Radul, Jonathan Rees, Zane Shelby, and Joshua Thayer for their help on earlier versions of this work. We have also benefited greatly from conversations with and feedback from Tan Zhi-Xuan, Nishad Gothoskar, McCoy Becker, Mathieu Huot, Feras Saad, Joshua Tenenbaum, Eli Sennesh, Jan-Willem van de Meent, Lionel Wong, Matt Bowers, Tony Chen, Tracey Mills, Cameron Freer, Jean-Baptiste Tristan, and Jesse Michel. Our anonymous referees also contributed very helpful feedback. This material is based on work supported by the NSF Graduate Research Fellowship under Grant No. 1745302. Our work is also supported under the DARPA Machine Common Sense and SAIL-ON projects.

REFERENCES

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. 2010. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72, 3 (2010), 269–342.

Christophe Andrieu and Gareth O Roberts. 2009. The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics* 37, 2 (2009), 697–725.

Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, et al. 2019. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–24.

Mark A Beaumont. 2003. Estimation of population growth or decline in genetically monitored populations. *Genetics* 164, 3 (2003), 1139–1160.

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. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978.

Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. 2015. The YCB object and model set: Towards common benchmarks for manipulation research. In *2015 International Conference on Advanced Robotics (ICAR)*. IEEE, 510–517.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017).

Yu-Hsi Cheng, Todd Millstein, Guy Van den Broeck, and Steven Holtzen. 2021. flip-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs. *arXiv preprint arXiv:2110.10284* (2021).

Nicolas Chopin, Omiros Papaspiliopoulos, et al. 2020. *An introduction to sequential Monte Carlo*. Springer.

Marco Cusumano-Towner and Vikash K Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. *Advances in Neural Information Processing Systems* 30 (2017).

Marco Francis Cusumano-Towner. 2020. Gen: A High-Level Programming Platform for Probabilistic Inference. 2011 (2020), 231.

Marco F Cusumano-Towner and Vikash K Mansinghka. 2018. Using probabilistic programs as proposals. *arXiv preprint arXiv:1801.03612* (2018).

Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. 2017. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977* (2017).

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*. 221–236.

Arnaud Doucet, Michael K Pitt, George Deligiannidis, and Robert Kohn. 2015. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. *Biometrika* 102, 2 (2015), 295–313.

Paul Fearnhead, Omiros Papaspiliopoulos, Gareth O Roberts, and Andrew Stuart. 2010. Random-weight particle filtering of continuous time processes. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72, 4 (2010), 497–512.

Martin A Fischler and Robert C Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* 24, 6 (1981), 381–395.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain (Proceedings of Machine Learning Research, Vol. 84)*, Amos J. Storkey and Fernando Pérez-Cruz (Eds.). PMLR, 1682–1690. http://proceedings.mlr.press/v84/ge18b.html

Timon Gehr, Samuel Steffen, and Martin T. Vechev. 2020. λPSI: exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 883–897. https://doi.org/10.1145/3385412.3386006

Noah D Goodman and Michael C Frank. 2016. Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences* 20, 11 (2016), 818–829.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Kallista A. Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, David A. McAllester and Petri Myllymäki (Eds.). AUAI Press, 220–229. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24

Nishad Gothoskar, Marco F. Cusumano-Towner, Ben Zinberg, Matin Ghavamizadeh, Falk Pollok, Austin Garrett, Josh Tenenbaum, Dan Gutfreund, and Vikash K. Mansinghka. 2021. 3DP3: 3D Scene Perception via Probabilistic Programming. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing*

Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 9600–9612. https://proceedings.neurips.cc/paper/2021/hash/4fc66104f8ada6257fa55f29a2a567c7-Abstract.html

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12.

Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 140:1–140:31. https://doi.org/10.1145/3428208

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image Generation From Scene Graphs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 1219–1228. https://doi.org/10.1109/CVPR.2018.00133

Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. 2015. Image retrieval using scene graphs. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 3668–3678. https://doi.org/10.1109/CVPR.2015.7298990

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards verified stochastic variational inference for probabilistic programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 16:1–16:33. https://doi.org/10.1145/3371084

Alexander K. Lew, Monica Agrawal, David A. Sontag, and Vikash Mansinghka. 2021. PClean: Bayesian Data Cleaning at Scale with Domain-Specific Probabilistic Programming. In *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 130)*, Arindam Banerjee and Kenji Fukumizu (Eds.). PMLR, 1927–1935. http://proceedings.mlr.press/v130/lew21a.html

Alexander K. Lew, Marco F. Cusumano-Towner, and Vikash K. Mansinghka. 2022. Recursive Monte Carlo and variational inference with auxiliary variables. In *Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1-5 August 2022, Eindhoven, The Netherlands (Proceedings of Machine Learning Research, Vol. 180)*, James Cussens and Kun Zhang (Eds.). PMLR, 1096–1106. https://proceedings.mlr.press/v180/lew22a.html

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. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 1–32. https://doi.org/10.1145/3371087

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proc. ACM Program. Lang.* 7, POPL (2023), 121–153. https://doi.org/10.1145/3571198

Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-Preserving, Dependence-Aware Guide Generation for Sound, Effective Amortized Probabilistic Inference. *Proc. ACM Program. Lang.* 7, POPL (2023), 1454–1482. https://doi.org/10.1145/3571243

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

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)*, Vol. 14. ACM, New York, NY, USA, 603–616. https://doi.org/10.1145/3192366.3192409

Eric Mays, Fred J Damerau, and Robert L Mercer. 1991. Context based spelling correction. *Information Processing & Management* 27, 5 (1991), 517–522.

Praveen Narayanan and Chung Chieh Shan. 2020. Symbolic Disintegration with a Variety of Base Measures. *ACM Transactions on Programming Languages and Systems* 42, 2 (2020). https://doi.org/10.1145/3374208

Radford M Neal. 2000. Markov chain sampling methods for Dirichlet process mixture models. *Journal of computational and graphical statistics* 9, 2 (2000), 249–265.

Tom Rainforth. 2018. Nesting Probabilistic Programs. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, Amir Globerson and Ricardo Silva (Eds.). AUAI Press, 249–258. http://auai.org/uai2018/proceedings/papers/92.pdf

Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B Schön, and David Broman. 2021. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications biology* 4, 1 (2021), 244.

Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic programming with fast exact symbolic inference. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2021), 804–819. https://doi.org/10.1145/3453483.3454078 arXiv:2010.03485

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational validation of higher-order Bayesian inference. *Proc. ACM Program. Lang.* 2, POPL (2018), 60:1–60:29. https://doi.org/10.1145/3158148

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. *Principles of Programming Languages* (2017). https://doi.org/10.1145/3009837.3009852

Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning proposals for probabilistic programs with inference combinators. In *Uncertainty in Artificial Intelligence*. PMLR, 1056–1066.

Andreas Stuhlmüller and Noah D. Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. In *2nd International Workshop on Statistical Relational AI (StaRAI-12), held at the Uncertainty in Artificial Intelligence Conference (UAI 2012), Catalina Island, CA, USA, August 18, 2012*, Henry A. Kautz, Kristian Kersting, Sriraam Natarajan, and David Poole (Eds.). https://starai.cs.kuleuven.be/2012/accepted/stuhlmuller.pdf

Minh-Ngoc Tran, Marcel Scharth, Michael K Pitt, and Robert Kohn. 2013. Importance sampling squared for Bayesian inference in latent variable models. *arXiv preprint arXiv:1309.3339* (2013).

Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2021), 788–803. https://doi.org/10.1145/3453483.3454077 arXiv:2104.03598

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014 (JMLR Workshop and Conference Proceedings, Vol. 33)*. JMLR.org, 1024–1032. http://proceedings.mlr.press/v33/wood14.html

Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–28.

Tan Zhi-Xuan, Jordyn L. Mann, Tom Silver, Joshua B. Tenenbaum, and Vikash K. Mansinghka. 2020. Online Bayesian goal inference for boundedly-rational planning agents. *Advances in Neural Information Processing Systems* 2020-December (2020). arXiv:2006.07532

Robert Zinkov and Chung-chieh Shan. 2016. Composing inference algorithms as program transformations. *arXiv preprint arXiv:1603.01882* (2016).

Matt Zucker, James Kuffner, and Michael Branicky. 2007. Multipartite RRTs for rapid replanning in dynamic environments. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, 1603–1609.

Received 2022-11-10; accepted 2023-03-31