Semantics of Probabilistic Program Traces

Alexander K. Lew
MIT
alexlew@mit.edu

Eli Sennesh
Northeastern University
esennesh@ccs.neu.edu

Jan-Willem van de Meent
University of Amsterdam
j.w.vandemeent@uva.nl

Vikash K. Mansinghka
MIT
vkm@mit.edu

1 Introduction

Reifying random choices that probabilistic programs make into *traces* has long been an important technique for both implementors [13, 14, 20, 21] and theorists [2, 16] of PPLs. But recently, many languages [1, 3, 4, 10, 17] have made traces a *user-facing* concern, rather than an implementation detail: users label each random sample in their programs with a *name*, and traces are *dictionaries* recording the names and values of all random choices encountered during program execution. Traces are often meant to be inspected, and explicitly reasoned about; for example, when invoking inference algorithms, users may have to ensure certain properties of the *distributions over traces* their programs encode, a task for which several static analyses have been proposed [7–9, 18]. In developing these analyses, researchers have defined restricted formal calculi for modeling aspects of modern, trace-based languages. But these calculi are crafted specially to model only the details that are relevant for a particular analysis, and as such, feature restrictions that make them unsuitable for "off-the-shelf" reuse when reasoning about new analyses or program transformations. For the same reason, they also fail to capture some interesting aspects of real-world PPLs.

This extended abstract presents a minimal language (Fig. 1) modeling modern, trace-based PPLs, and a denotational semantics that assigns to each program an s-finite measure over a space of high-level traces (Fig. 2). It then demonstrates how to use this formal calculus to define program transformations and prove them sound, using as examples *weighted sampling* and *density evaluation* (Figs. 3 and 4). Finally, it shows how to reason about higher-level inference algorithms that compose these program transformations, via the example of importance sampling with a custom proposal (Fig. 5). We aim to: (1) give theorists a self-contained model of a typical trace-based PPL, (2) give PPL developers a blueprint for formal reasoning about program semantics & new program transformations, and (3) provide a common reference point for discussion and comparison of many real-world PPLs.

2 Denotational Semantics

Our language is a simply-typed λ-calculus (Fig. 1) extended with minimal constructs for building traced probabilistic programs: $\mathbf{sample}(\mathit{dist}, \mathit{name})$ for drawing a named sample from a primitive distribution, and $\mathbf{factor}(w)$ for factoring a non-negative number w into the likelihood. Our semantics

$$\begin{aligned}
\text{Ground types } \sigma &::= 1 \mid \mathbb{B} \mid \mathbb{R} \mid \mathrm{Str} \mid \mathrm{Trace} \mid \sigma_1 \times \sigma_2 \\
\text{Types } \tau &::= \sigma \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid D\,\sigma \mid M\,\tau \\
\text{Terms } t &::= () \mid c \mid x \mid \lambda x.t \mid t_1\, t_2 \mid \{\} \mid \{t_1 \mapsto t_2\} \mid \\
&\quad\ \mathbf{sample}(t_1, t_2) \mid \mathbf{factor}\, t \mid \mathbf{return}\, t \mid \mathbf{do}\{m\} \\
\text{Do-notation } m &::= t \mid x \leftarrow t; m
\end{aligned}$$

$$\frac{t_1 : \mathrm{Str} \quad t_2 : \sigma}{\{t_1 \mapsto t_2\} : \mathrm{Trace}} \qquad \frac{t : \tau}{\mathbf{return}\, t : M\,\tau} \qquad \frac{t_1 : D\,\sigma \quad t_2 : \mathrm{Str}}{\mathbf{sample}(t_1, t_2) : M\,\sigma}$$

$$\frac{t : \mathbb{R}}{\mathbf{factor}\, t : M\,1} \qquad \frac{t : M\,\tau}{\mathbf{do}\{t\} : M\,\tau} \qquad \frac{t : M\,\tau_1 \quad x : \tau_1 \vdash \mathbf{do}\{m\} : M\,\tau_2}{\mathbf{do}\{x \leftarrow t; m\} : M\,\tau_2}$$

Figure 1. Grammar and selected typing rules of our calculus. The symbol c ranges over *constants* of any type, including ground types (e.g., **false** or 3.14), distribution types (e.g., primitives for Gaussian and Bernoulli distributions), and function types (built-in operations, including projections π_1, π_2 for pairs, and $\mathbf{if}_\tau : \mathbb{B} \to (1 \to \tau) \to (1 \to \tau) \to \tau$). We write $\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$ as sugar for $(\lambda x.t_1)\ t_2$.

Semantics of types (quasi-Borel spaces) $\llbracket \mathbb{B} \rrbracket := \mathbb{B}$ $\quad \llbracket 1 \rrbracket := 1$

$\llbracket D\,\sigma \rrbracket := \mathcal{M}\,\llbracket \sigma \rrbracket \quad \llbracket M\,\tau \rrbracket := \mathcal{M}\,\mathbb{T} \times (\mathbb{T} \to \llbracket \tau \rrbracket) \quad \llbracket \mathbb{R} \rrbracket := \mathbb{R} \quad \llbracket \mathrm{Str} \rrbracket := \mathrm{Str}$

$\llbracket \mathrm{Trace} \rrbracket := \mathbb{T} \quad \llbracket \tau_1 \times \tau_2 \rrbracket := \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \quad \llbracket \tau_1 \to \tau_2 \rrbracket := \llbracket \tau_1 \rrbracket \to \llbracket \tau_2 \rrbracket$

Semantics of terms (quasi-Borel functions mapping environments to values)

$\llbracket c \rrbracket(\gamma) := \underline{c} \quad \llbracket x \rrbracket(\gamma) := \gamma[x] \quad \llbracket \lambda x.t \rrbracket(\gamma) := \lambda v.\llbracket t \rrbracket(\gamma[x \mapsto v])$

$\llbracket t_1\, t_2 \rrbracket(\gamma) := \llbracket t_1 \rrbracket(\gamma)\,\llbracket t_2 \rrbracket(\gamma) \quad \llbracket \mathbf{return}\, t \rrbracket(\gamma) := (\delta_{\{\}}, \lambda u.\llbracket t \rrbracket(\gamma))$

$$\llbracket \mathbf{sample}(t_1, t_2) \rrbracket(\gamma) := \Big(\int \delta_{\{\llbracket t_2 \rrbracket(\gamma) \mapsto x\}} \llbracket t_1 \rrbracket(\gamma, dx), \lambda u.\pi_1(\underline{\mathsf{pop}}\, u\, \llbracket t_2 \rrbracket(\gamma))\Big)$$

$\llbracket \mathbf{factor}\ t \rrbracket(\gamma) := (e^{\llbracket t \rrbracket(\gamma)} \odot \delta_{\{\}}, \lambda u.()) \quad \llbracket \mathbf{do}\{t\} \rrbracket(\gamma) := \llbracket t \rrbracket(\gamma)$

$$\begin{aligned}
&\llbracket \mathbf{do}\{x \leftarrow t; m\} \rrbracket(\gamma) := \\
&\Bigg(\iint (\underline{\mathtt{disj}}(u, v) \odot \delta_{u +\!\!+ v})(\pi_1 \circ \llbracket \mathbf{do}\{m\} \rrbracket)(\gamma[x \mapsto \pi_2(\llbracket t \rrbracket(\gamma))(u)], dv) \\
&\qquad\qquad (\pi_1 \circ \llbracket t \rrbracket)(\gamma, du), \\
&\quad \lambda u.\pi_2(\llbracket \mathbf{do}\{m\} \rrbracket(\gamma[x \mapsto \pi_2(\llbracket t \rrbracket(\gamma))(u)]))(u)\Bigg)
\end{aligned}$$

Figure 2. Our semantics, interpreting types as quasi-Borel spaces and terms as quasi-Borel functions. Primitive distributions $D\,\sigma$ are intrepreted as measures over the space denoted by σ ($\mathcal{M}\,\llbracket \sigma \rrbracket$), whereas compound probabilistic programs $M\,\tau$ are interpreted as *pairing* a measure over traces ($\mathcal{M}\,\mathbb{T}$) with a "value function" mapping traces to outputs ($\mathbb{T} \to \tau$).

LAFI 2023, January 2023, Boston, MA, USA
2023.

(Fig. 2) interprets each type τ in the language as a *quasi-Borel space* $[\![\tau]\!]$ (an alternative to *measurable spaces* suitable for higher-order PPL semantics [5]). We highlight several key points, and refer the reader to Appendix B for full details:

- A novel contribution is a quasi-Borel space $\mathbb{T}$ of traces. Previous work has modeled traces as lists [16] or trees [13] of reals, recording primitive uniform draws. But to model the density calculations in many PPLs, traces must record the *heterogeneous* values returned by diverse primitive distributions, e.g. Bernoulli booleans and Gaussian reals. Our traces are dictionaries mapping string names to heterogeneous values of ground type. The syntax $\{\}$ builds an empty trace, $\{t_1 \mapsto t_2\}$ builds a trace mapping name t_1 to value t_2, and $\mathtt{concat}\ t_1\ t_2$ (which we abbreviate $t_1 +\!\!+ t_2$) concatenates two traces (or returns the empty trace if names overlap). The primitive $\mathsf{pop}_\sigma\ t_1\ t_2$ looks up the name t_2 in the trace t_1, and if it finds a value of type σ, returns it, and a trace containing the remainder of the entries; otherwise, it returns a *default value* of type σ, and the empty trace.
- The type $M\,\tau$ of probabilistic programs is interpreted in the quasi-Borel space $\mathcal{M}\,\mathbb{T} \times (\mathbb{T} \to [\![\tau]\!])$: a program denotes both a quasi-Borel measure over traces, and a function from traces to return values. The probabilistic program **return** t that deterministically computes the term t denotes a Dirac distribution over the empty trace $\{\}$, together with a function mapping the empty trace to $[\![t]\!]$. The program $\mathbf{sample}(t_1, t_2)$ denotes a probability distribution over singleton traces, and when programs are sequenced using **do**, their traces concatenate. Programs that use a name twice denote the zero measure.

3 Program Transformations

PPLs typically automate core operations that form a *computational interface* to the measure a program denotes, by transforming the program or interpreting it in a non-standard way [19]. Figs. 3 and 4 show transformations implementing the following simple interface, for a measure μ over traces:

- **Density evaluation:** given a trace u, evaluate $\frac{d\mu}{d\mathbb{B}_{\text{Trace}}}(u)$, where $\mathbb{B}_{\text{Trace}}$ is a *base measure* over $\mathbb{T}$ (Appendix C).
- **Weighted sampling:** sample a pair (u, w) of a trace and weight s.t. $\mathbb{E}[w \cdot 1_A(u)] = \mu(A)$ for measurable $A \subseteq \mathbb{T}$.

Similar transformations are well-known in the PPL community, and can be validated using various proof techniques, including via Ścibior et al. [16]'s framework for "inference transformations," or by modeling them with algebraic effects [11, 12, 15]. In Appendix C, we prove the following correctness result using logical relations:

Proposition 3.1. *Let $(\mu, f) = [\![p]\!]$ for some $\vdash p : M\,\tau$. Then:*

- *$[\![density\{p\}]\!]$ is a density of μ with respect to $\mathbb{B}_{Trace}$, and*
- *$\mathcal{V}[\![wsamp\{p\}]\!]$ (where $\mathcal{V}(\mu, f) = f_*\mu$) is a probability measure, and $\iint (w \odot \delta_u)\ (\mathcal{V}[\![wsamp\{p\}]\!])(d(u, w)) = \mu$.*

Top-level wrapper $(\vdash t : M\,\tau \implies \vdash density\{t\} : \text{Trace} \to \mathbb{R})$

$$density\{t\} := \lambda\hat{u}.\mathbf{let}\ (w, v, u') = \rho\{t\}(\hat{u})\ \mathbf{in}\ \mathtt{isempty}(u') \cdot w$$

Transforming types (identity on ground types σ)

$$\rho\{D\,\sigma\} := \sigma \to \mathbb{R} \quad \rho\{M\,\tau\} := \text{Trace} \to \mathbb{R} \times \rho\{\tau\} \times \text{Trace}$$
$$\rho\{\tau_1 \times \tau_2\} := \rho\{\tau_1\} \times \rho\{\tau_2\} \quad \rho\{\tau_1 \to \tau_2\} := \rho\{\tau_1\} \to \rho\{\tau_2\}$$

Transforming terms

$$\rho\{c\} := c_\rho \quad \rho\{x\} := x \quad \rho\{\lambda x.t\} := \lambda x.\rho\{t\}$$
$$\rho\{\mathbf{sample}(t_1, t_2)\} := \lambda\hat{u}.\mathbf{let}\ (\hat{v}, \hat{u}') = \mathsf{pop}_\sigma\ \hat{u}\ t_2\ \mathbf{in} \quad [\textit{when } t_1 : D\,\sigma]$$
$$(\mathsf{has}_\sigma\ \hat{u}\ t_2 \cdot \rho\{t_1\}(\hat{v}), \hat{v}, \hat{u}')$$
$$\rho\{\mathbf{factor}\ t\} := \lambda\hat{u}.(\exp(t), (), \hat{u}) \quad \rho\{t_1\,t_2\} := \rho\{t_1\}\,\rho\{t_2\}$$
$$\rho\{\mathbf{do}\{t\}\} := \rho\{t\} \quad \rho\{\mathbf{return}\,t\} := \lambda\hat{u}.(1, \rho\{t\}, \hat{u})$$
$$\rho\{\mathbf{do}\{x \leftarrow t; m\}\} := \lambda\hat{u}.\mathbf{let}\ (\hat{w}, x, \hat{u}') = \rho\{t\}(\hat{u})\ \mathbf{in}$$
$$\mathbf{let}\ (\hat{v}, \hat{r}, \hat{u}'') = \rho\{\mathbf{do}\{m\}\}(\hat{u}')\ \mathbf{in}$$
$$(\hat{w} \cdot \hat{v}, \hat{r}, \hat{u}'')$$

Figure 3. Trace density program transformation.

Top-level wrapper $(\vdash t : M\,\tau \implies \vdash wsamp\{t\} : M\,(\text{Trace} \times \mathbb{R}))$

$$wsamp\{t\} := \mathbf{do}\{(u, w, v) \leftarrow \omega\{t\}; \mathbf{return}\ (u, w)\}$$

Transforming types (identity on ground types σ)

$$\omega\{D\,\sigma\} := D\,\sigma \quad \omega\{M\,\tau\} := M\,(\text{Trace} \times \mathbb{R} \times \omega\{\tau\})$$
$$\omega\{\tau_1 \times \tau_2\} := \omega\{\tau_1\} \times \omega\{\tau_2\} \quad \omega\{\tau_1 \to \tau_2\} := \omega\{\tau_1\} \to \omega\{\tau_2\}$$

Transforming terms

$$\omega\{c\} := c_\omega \quad \omega\{x\} := x \quad \omega\{\lambda x.t\} := \lambda x.\omega\{t\}$$
$$\omega\{t_1\,t_2\} := \omega\{t_1\}\,\omega\{t_2\} \quad \omega\{\mathbf{factor}\ t\} := \mathbf{return}(\{\}, \exp(t), ())$$
$$\omega\{\mathbf{sample}(t_1, t_2)\} := \mathbf{do}\{x \leftarrow \mathbf{sample}(t_1, t_2); \mathbf{return}\ (\{t_2 \mapsto x\}, 1, x)\}$$
$$\omega\{\mathbf{do}\{t\}\} := \omega\{t\} \quad \omega\{\mathbf{return}\,t\} := \mathbf{return}(\{\}, 1, \omega\{t\})$$
$$\omega\{\mathbf{do}\{x \leftarrow t; m\}\} := \mathbf{do}\{(\hat{t}, \hat{w}, x) \leftarrow \omega\{t\}; (\hat{s}, \hat{v}, \hat{y}) \leftarrow \omega\{\mathbf{do}\{m\}\};$$
$$\mathbf{return}\ (\hat{t} +\!\!+ \hat{s}, \hat{w} \cdot \hat{v} \cdot \mathtt{disj}(\hat{t}, \hat{s}), \hat{y})\}$$

Figure 4. Weighted trace sampler program transformation.

$$\begin{aligned} importance\{p, q\} := \mathbf{do}\{(\hat{t}, \hat{w}) &\leftarrow wsamp\{q\}; \\ &\mathbf{let}\ \hat{w}_q = density\{q\}(\hat{t})\ \mathbf{in} \\ &\mathbf{let}\ \hat{w}_p = density\{p\}(\hat{t})\ \mathbf{in} \\ &\mathbf{return}\ (\hat{t}, \hat{w} \cdot \frac{\hat{w}_p}{\hat{w}_q})\} \end{aligned}$$

Figure 5. Importance sampling with a custom proposal

4 Sound Inference

Inference algorithms can be built using the operations presented in Sec. 3. For example, Fig. 5 implements an importance sampler targeting p but using q's sampler as a proposal:

Proposition 4.1. *Let $\vdash p, q : M\tau$, let μ_p, μ_q be the measures they denote, and let ν be $\mathcal{V}[\![importance\{p, q\}]\!]$, a probability measure over $\mathbb{T} \times \mathbb{R}_{\geq 0}$. If $\mu_q \neq 0$ and $\mu_p \ll \mu_q$, then $\pi_{1*}\nu = \pi_{1*}(\mathcal{V}[\![wsamp\{q\}]\!])$, and $\iint (\pi_2(x) \odot \delta_{\pi_1(x)})\nu(dx) = \mu_p$.*

5 Discussion

An interesting direction for future work, which we sketch in Appendix E, is to understand today's trace-based PPL landscape in terms of how each PPL extends Fig. 1's language, and what computational interface it exposes to the measures denoted by programs. The choice of computational interface affects both the modeling constructs the PPL *can* expose (e.g., we could not expose a tractable *density* operation if our language featured a **normalize** construct), and the expressible inference algorithms (e.g., HMC cannot be implemented against Sec. 3's interface, as it requires gradients).

References

[1] 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.

[2] Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices* 51, 9 (2016), 33–46.

[3] 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.

[4] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*. PMLR, 1682–1690.

[5] 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.

[6] Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing.. In *FoSSaCS*. 319–338.

[7] Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2022. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *arXiv preprint arXiv:2208.10530* (2022).

[8] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–33.

[9] Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

[10] 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*. 603–616.

[11] Dave Moore and Maria I Gorinova. 2018. Effect handling for composable program transformations in edward2. *arXiv preprint arXiv:1811.06150* (2018).

[12] Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu. 2022. Modular Probabilistic Models via Algebraic Effects. *arXiv preprint arXiv:2203.04608* (2022).

[13] Hugo Paquet and Sam Staton. 2021. LazyPPL: laziness and types in non-parametric probabilistic programs. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*.

[14] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. 165–176.

[15] Adam Scibior and Ohad Kammar. 2015. Effects in Bayesian inference. In *Workshop on Higher-Order Programming with Effects (HOPE)*.

[16] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational validation of higher-order Bayesian inference. *arXiv preprint arXiv:1711.03219* (2017).

[17] 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.

[18] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 788–803.

[19] David Wingate, Noah Goodman, Andreas Stuhlmüller, and Jeffrey Siskind. 2011. Nonstandard interpretations of probabilistic programs for efficient inference. *Advances in neural information processing systems* 24 (2011).

[20] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 770–778.

[21] Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. 2014. Generating efficient MCMC kernels from probabilistic programs. In *Artificial Intelligence and Statistics*. PMLR, 1068–1076.

Appendix

A Background and Notation

Quasi-Borel Spaces. We carry out our development in the category of quasi-Borel spaces [5], an alternative to measurable spaces which we review briefly here.

Definition A.1 (quasi-Borel space). A *quasi-Borel space* X is a tuple $(|X|, M_X)$ of a carrier set $|X|$ and a set $M_X \subseteq [\mathbb{R} \to |X|]$ of *admissible random elements*, satisfying:

- **(Closure under measurable precomposition.)** If $\phi \in M_X$ and $f : \mathbb{R} \to \mathbb{R}$ is measurable, $\phi \circ f \in M_X$.
- **(Constant maps.)** If $\phi = \lambda r.x$ for some $x \in |X|$, then $\phi \in M_X$.
- **(Closure under piecewise gluing.)** If $\{A_i\}_{i \in \mathbb{N}}$ is a countable partition of $\mathbb{R}$ and there exist $\phi_i \in M_X$ such that $\phi(r) = \phi_i(r)$ whenever $r \in A_i$, then $\phi \in M_X$.

Definition A.2 (quasi-Borel function). If X and Y are quasi-Borel spaces, a *quasi-Borel function* $f : X \to Y$ is a function from $|X|$ to $|Y|$ satisfying the property that for all $\phi \in M_X$, $f \circ \phi \in M_Y$.

Example A.3 (Standard Borel Spaces). Any standard Borel space (measurable space that is either finite, countable, or measurably isomorphic to $\mathbb{R}$) is also a quasi-Borel space, by choosing M_X to be the measurable functions from $\mathbb{R}$ to X. The quasi-Borel functions between standard Borel spaces are exactly the measurable functions between them as measurable spaces.

Definition A.4 (quasi-Borel measure). Let X be a quasi-Borel space. A *quasi-Borel measure on* X is an equivalence class of (μ, α) pairs, where μ is a (measure-theoretic) measure on $\mathbb{R}$, and $\alpha \in M_X$. Two pairs (μ_1, α_1) and (μ_2, α_2) are equivalent if for all quasi-Borel $f : X \to \mathbb{R}_{\geq 0}$, $\int f(\alpha_1(x))\mu_1(dx) = \int f(\alpha_2(x))\mu_2(dx)$.

Proposition A.5. *There is a strong commutative monad $\mathcal{M}$ in QBS, taking X to the space $\mathcal{M}\,X$ of measures on X.*

Notation. Following Ścibior et al. [16], we use *synthetic measure theory* notation to concisely write down quasi-Borel measures:

- For $x \in |X|$, we write δ_x for the Dirac delta distribution at x. (We can represent it as a quasi-Borel measure by taking $\mu = U(0, 1)$ and $\alpha = \lambda r.x$.)
- For $\mu \in |\mathcal{M}\,X|$, and $w : X \to \mathbb{R}_{\geq 0}$, we write $w \odot \mu$ for the measure with density w with respect to μ. For $w \in \mathbb{R}_{\geq 0}$, we abuse notation and write $w \odot \mu$ instead of $(\lambda x.w) \odot \mu$.
- For $f : X \to \mathbb{R}_{\geq 0}$ and $p = [(\mu, \alpha)] \in |\mathcal{M}\,X|$, we write $\int_X f(x)p(dx)$ for the integral $\int f(\alpha(r))\mu(dr)$.
- If $k : X \to \mathcal{M}\,Y$, then we write $k(x, dy)$ in integral expressions, rather than $k(x)(dy)$.
- For $k : X \to \mathcal{M}\,Y$, we write $⨏_X k(x)p(dx)$ for the measure over Y that integrates a function $g : Y \to \mathbb{R}_{\geq 0}$ by computing $\iint g(y)p(dx)k(x, dy)$.
- For $f : X \to Y$, and $\mu : \mathcal{M}\,X$, $f_*\mu : \mathcal{M}\,Y$ is the pushforward of μ by f, integrating a function $g : Y \to \mathbb{R}_{\geq 0}$ by integrating $g \circ f$ under μ.

B Core Calculus Syntax and Semantics: Details

Language syntax: types. Our language (Fig. 1) has as ground types 1 (the singleton type), the Booleans $\mathbb{B}$, the reals $\mathbb{R}$, the strings Str, tuples $\sigma_1 \times \sigma_2$ of other ground types, and a new type of *traces*, which we discuss below. In addition, the language features function types $\tau_1 \to \tau_2$, types $D\,\sigma$ of primitive distributions over each ground type σ, and monadic types $M\,\tau$ representing "traced probabilistic computations returning τ."

Language syntax: terms. Our language's terms include *constants c* of various types (e.g., **false** is a Boolean constant, 3.14 is a real-valued constant, **normal** is a constant of type $\mathbb{R} \times \mathbb{R} \to D\,\mathbb{R}$, and + is a constant of type $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$), variables x, function terms $\lambda x.t$, and function application expressions $t_1\ t_2$ (where t_1 has function type, and t_2 is its argument). We also include syntax for constructing empty ($\{\}$) and singleton ($\{name \mapsto value\}$) traces, and a standard **let** expression for defining local variables. For probabilistic programming, **return** t constructs a deterministic probabilistic program of type $M\,\tau$ that just computes a value of type τ; **sample**(t_1, t_2) is the program that samples from distribution t_1 at name t_2; **factor**(t) factors a non-negative real weight into the likelihood, creating a possibly unnormalized probabilistic program; and **do**$\{x \leftarrow t_1; m\}$ is used to build larger probabilistic computations, that first run a computation t_1, assigning x to the result, then run the remainder of the computation m. (Our syntax is inspired by Haskell's **do**-notation for sequencing monadic computations.)

Semantics of types. Our semantics (Fig. 2) interprets each type τ as a quasi-Borel space $[\![\tau]\!]$. Our ground types are all interpreted as *standard Borel spaces*, measurable spaces that are either finite, countable, or isomorphic to $\mathbb{R}$, and as such extend canonically to quasi-Borel spaces: $[\![\mathbb{R}]\!] = (\mathbb{R}, \mathcal{B}(\mathbb{R}))$, $[\![\text{Str}]\!] = (\text{Str}, \mathcal{P}(\text{Str}))$, $[\![\mathbb{B}]\!] = (\{\textbf{True}, \textbf{False}\}, \mathcal{P}(\{\textbf{True}, \textbf{False}\}))$,

$[\![1]\!] = (\{()\}, \mathcal{P}(\{()\}))$, and $[\![\sigma_1 \times \sigma_2]\!] = [\![\sigma_1]\!] \times [\![\sigma_2]\!]$. For the ground type Trace, we define a new standard Borel space, $\mathbb{T}$—see below. Our semantics interprets function types as quasi-Borel function spaces (just as in Ścibior et al. [16]), and for the types $D\,\sigma$ of distributions over ground types, we have $[\![D\,\sigma]\!] = \mathcal{M}\,[\![\sigma]\!]$, the quasi-Borel space of measures over $[\![\sigma]\!]$. Because $[\![\sigma]\!]$ is always standard Borel, these are just the ordinary (measure-theoretic) measures over $[\![\sigma]\!]$.

Importantly, our semantics interprets monadic probabilistic programs $M\,\tau$, not as (directly) denoting quasi-Borel measures over $[\![\tau]\!]$, but rather as denoting *pairs* containing a quasi-Borel measure over traces, and a $\mathbb{T} \to [\![\tau]\!]$ function mapping traces to return values. If t is a term of type $M\,\tau$, we write $\mathcal{V}[\![t]\!]$ for the marginal measure over τ that "forgets the trace," $\mathcal{V}[\![t]\!] = \pi_2([\![t]\!])_*(\pi_1([\![t]\!]))$. However, a key thesis of this work is that it is useful to keep around the specific distribution over traces implemented by the user's program, and not treat all programs with equivalent marginal output distributions as equivalent.

QBS $\mathbb{T}$ of traces. We define a measurable space of traces that is standard Borel, and thus is also a quasi-Borel space. First, we define a set $\mathbb{S}$ of *trace shapes*, lexicographically sorted lists of (k, σ) pairs, where $k \in$ Str is a string-valued name, σ is a ground type, and no name appears more than once in the list. Then, for $i \geq 0$, we define $\mathbb{T}_i = \{(s, v) \mid s \in \mathbb{S} \wedge v \in \bigtimes_{(k,\sigma)\in s}[\![\sigma]\!]_i\}$, where $[\![\cdot]\!]_i$ is an inductively defined family of (SBS-valued) semantic functions: we have $[\![\mathbb{R}]\!]_i = (\mathbb{R}, \mathcal{B}(\mathbb{R}))$, $[\![\mathrm{Str}]\!]_i = (\mathrm{Str}, \mathcal{P}(\mathrm{Str}))$, $[\![\mathbb{B}]\!]_i = (\{\mathbf{True}, \mathbf{False}\}, \mathcal{P}(\{\mathbf{True}, \mathbf{False}\}))$, $[\![1]\!]_i = (\{()\}, \mathcal{P}(\{()\}))$, $[\![\sigma_1 \times \sigma_2]\!]_i = [\![\sigma_1]\!]_i \times [\![\sigma_2]\!]_i$, $[\![\mathrm{Trace}]\!]_0 = \emptyset$ (the empty set, with the σ-algebra $\{\emptyset\}$), and $[\![\mathrm{Trace}]\!]_i = \mathbb{T}_{i-1}$, with the σ-algebra that makes $U \subseteq \mathbb{T}_{i-1}$ measurable if for each $s \in \mathbb{S}$, $\{v \mid (s, v) \in U\}$ is measurable under the product σ-algebra for $\bigtimes_{k,\sigma\in s}[\![\sigma]\!]_{i-1}$. We then set $\mathbb{T} = \cup_{i\in\mathbb{N}}\mathbb{T}_i$, with the σ-algebra that makes $U \subseteq \mathbb{T}$ measurable if for each $i \in \mathbb{N}$, $\{u \in U \mid u \in \mathbb{T}_i \wedge \forall j < i, u \notin \mathbb{T}_j\}$ is measurable as a subset of $\mathbb{T}_i$.[1]

We expose several primitive functions for dealing with traces:

- has_σ : Trace $\to$ Str $\to \mathbb{R}$: returns 1 if the given trace has a value of type σ at the given name, 0 otherwise.
- pop_σ : Trace $\to$ Str $\to \sigma \times$ Trace: if the given trace has a value of type σ at the given name, return it, along with a modified trace that deletes that entry. Otherwise, return a *default value* of type σ (0 for $\mathbb{R}$, **false** for $\mathbb{B}$, "" for Str, {} for Trace, () for 1, and pairs of default values for tuples), and an empty trace.

The names has_σ and pop_σ are syntax in our language; we write $\underline{c}$ to refer to the quasi-Borel functions they denote.

Semantics of terms. Fig. 2 also defines the semantics function $[\![\cdot]\!]$ on terms. Technically, the domain of the $[\![\cdot]\!]$ function is *typed terms-in-context* of the form $\Gamma \vdash t : \tau$, where Γ is a list of $x_i : \tau_i$ entries for each free variable in t, t is a well-formed term in context Γ, and τ is the type of t in context Γ. However, for brevity we omit Γ and τ, writing $[\![t]\!]$. The denotation of a term in context is a quasi-Borel function, from *environments* γ assigning values to each free variable in Γ, to values in $[\![\tau]\!]$.

For the deterministic portion of our language, our semantics is standard. The interesting cases are for terms of type $M\,\tau$, which have as denotations pairs of a measure over traces, and a value function mapping a trace to the expression's value under that trace. The terms of type $M\,\tau$ are:

- **return** t, which denotes a Dirac distribution over empty traces (because it makes no random choices), together with the value function that ignores the input trace and just returns $[\![t]\!](\gamma)$ (the value of t in the current environment γ).
- **sample**(t_1, t_2), which denotes the marginal distribution arising by sampling $x \sim [\![t_1]\!](\gamma)$ (recall that $t_1 : D\,\sigma$ denotes a primitive distribution), and returning the trace $\{[\![t_2]\!](\gamma) \mapsto x\}$. The return value function takes a trace u, and uses $\underline{\mathsf{pop}}_\sigma$ to look up the name $[\![t_2]\!](\gamma)$ in the trace, and return the value found there.
- **factor** t factors the exponent of a given log weight into the density. As such, its measure over traces is the dirac measure over {}, scaled by $e^{[\![t]\!](\gamma)}$. Its return value function trivially returns (), the empty tuple of type 1.
- **do**$\{x \leftarrow t; m\}$ sequences two probabilistic computations. The measure over traces that it denotes first generates a trace $u_1 \sim \pi_1([\![t]\!](\gamma))$, computes the return value $v = \pi_2([\![t]\!](\gamma))(u)$, then generates the rest of the trace $u_2 \sim \pi_1([\![\mathbf{do}\{m\}]\!](\gamma[x \mapsto v))$. If u_1 and u_2 do not have disjoint sets of trace names (which is checked by the `disj` primitive), the zero measure is returned, indicating error. Otherwise, the concatenation $u_1 +\!\!+\, u_2$ is returned. The value function passes a given trace into $[\![t]\!](\gamma)$'s return value function to get v, then into $[\![\mathbf{do}\{m\}]\!](\gamma[x \mapsto v])$'s return value function to get a final return value.

C Computational Interface: Details

Reasoning about the density transformation. First, in order to talk rigorously about densities, we need to define the *reference measures* with respect to which densities are computed:

[1]This inductive definition is designed to allow traces to contain other traces, which is one way of modeling *hierarchical* addresses like those used in Gen [3]. If only simple ground types (strings, reals, Booleans) were allowed, the inductive definition would not be required, and we could simply set $\mathbb{T} = \mathbb{T}_0$.

Definition C.1. For each ground type in our language, we assign a *base measure* $\mathbb{B}_\sigma$: for the reals we choose the Lebesgue measure, for 1, Str, and $\mathbb{B}$, we choose the counting measure, and for $\sigma_1 \times \sigma_2$, we choose the product of $\mathbb{B}_{\sigma_1}$ and $\mathbb{B}_{\sigma_2}$. For Trace, we first define base measures for the sets $\mathbb{T}_i$ considered in the previous section: in particular, we choose the measure that assigns to a subset $A \subseteq \mathbb{T}_i$ the measure $\sum_{s \in \mathbb{S}}(\bigtimes_{(k,\sigma)\in s} \mathbb{B}_\sigma)(\{v \mid (s,v) \in A\})$, where $\mathbb{B}_{\text{Trace}}$ is interpreted as the base measure for $\mathbb{T}_{i-1}$. Then, for the base measure over the space of all traces, for $A \subseteq \mathbb{T}$, we set $\mathbb{B}_{\text{Trace}}(A) = \sum_{i=0}^{\infty} \mathbb{B}_{\mathbb{T}_i}(\{t \in A \mid t \in \mathbb{T}_i \wedge \forall j < i, t \notin \mathbb{T}_j\})$.

Now, we can establish the correctness of the density program transformation given in Fig. 3. The main translation itself is one line: it translates a term $\vdash t : M\,\tau$ into a term $\vdash$ *density*$\{t\}$: Trace $\to \mathbb{R}$. But the top-level translation relies crucially on a "macro" ρ, that compositionally rewrites the program according to the rules in Fig. 3. The macro ρ operates on terms, but Fig. 3 also defines it on *types*; the invariant is that given a term of type τ, ρ will translate it into a term of type $\rho\{\tau\}$.

Logical relations for correctness of the density macro. What is the goal of this macro? We first clearly define a *specification* for ρ. For each type τ, we define a *relation* on $[\![\tau]\!] \times [\![\rho\{\tau\}]\!]$, which is meant to capture what it would mean for ρ to be doing its job correctly: if $s : \rho\{\tau\}$ is a correct translation of $t : \tau$, then $([\![t]\!], [\![s]\!])$ should be *related*. The relation is defined as follows:

- For ground types σ, $\mathcal{R}_\sigma := \{(x,x) \mid x \in [\![\sigma]\!]\}$. This reflects that the translation ρ does nothing to terms of ground type—a value is a 'correct translation' of another value only if the two values are exactly equal.
- For distribution types $D\,\sigma$, $\mathcal{R}_{D\,\sigma} := \{(d,p) \mid p \odot \mathbb{B}_\sigma = d\}$. In other words, a distribution $d \in [\![D\,\sigma]\!]$ is related to a function $p \in [\![\rho\{D\,\sigma\}]\!] = [\![\sigma \to \mathbb{R}]\!]$ if p is a correct density of d.
- For probabilistic program types $M\,\tau$, $\mathcal{R}_{M\,\tau} := \{((\mu, f), p) \mid (\underline{isempty} \circ \pi_3 \circ p) \odot ((\pi_1 \circ p) \odot \mathbb{B}_{\text{Trace}}) = \mu \wedge (f, \pi_2 \circ p) \in \mathcal{R}_\tau \wedge \forall u \in \{u \in \mathbb{T} \mid \pi_3(p(u)) = \{\}\}, \forall t \in \{t \in \mathbb{T} \mid \underline{disj}(u,t)\}, \pi_3(\underline{p(u +\!\!+ t)}) = t\}$. Unpacking this, we see that it is a relation between the denotations of probabilistic programs (containing a measure μ over traces and a function f that computes τ values based on traces) and their translations under ρ, into functions $p : \mathbb{T} \to \mathbb{R} \times [\![\rho\{\tau\}]\!] \times \mathbb{T}$. It places three requirements on the translation p:
  1. $(\underline{isempty} \circ \pi_3 \circ p) \odot ((\pi_1 \circ p) \odot \mathbb{B}_{\text{Trace}}) = \mu$, which says that the function $\lambda u.\underline{isempty}(\pi_3(p(u))) \cdot \pi_1(p(u))$ is a density of μ with respect to the base measure. Given a trace u, this density function runs $\underline{p(u)}$, extracts the first component (a real number), and multiplies it by 1 if the third return value from p is empty, or 0 otherwise. The idea is that p's *last* (third) return value is supposed to report any "unused" part of the input trace, and if this is non-empty, it means the input trace was "too big" to be part of μ's support. Therefore, the input trace has density 0.
  2. $(f, \pi_2 \circ p) \in \mathcal{R}_\tau$. This says that looking at just the second return value of p should basically implement the return-value function f. However, instead of saying that $f = \pi_2 \circ p$, we just require that f and $\pi_2 \circ p$ are *related*, i.e., that $\pi_2 \circ p$ is a correct translation of f. In the case where the value type τ is a ground type, "related" means "exactly equal." But if the original program was of type, e.g., $M\,(D\,\sigma)$ (a probabilistic program returning a primitive distribution), then the output type after translation is no longer $D\,\sigma$ but rather $\sigma \to \mathbb{R}$ (density functions of the primitive distribution). Because of this, f will map traces to $D\,\sigma$ values, whereas $\pi_2 \circ p$ will map traces to the densities of those primitive distributions.
  3. $\forall u \in \{u \in \mathbb{T} \mid \pi_3(p(u)) = \{\}\}, \forall t \in \{t \in \mathbb{T} \mid \underline{disj}(u,t)\}, \pi_3(p(u +\!\!+ t)) = t$. This requirement states that the third return value from p returns the "unconsumed" part of the input trace. More precisely, for any trace u on which p returns an empty "unconsumed" trace, if we extend u with extra choices t, we expect p's third return value to just be t.
- For product types $\tau_1 \times \tau_2$, $\mathcal{R}_{\tau_1 \times \tau_2} := \{((x,y),(x_\rho,y_\rho)) \mid (x,x_\rho) \in \mathcal{R}_{\tau_1} \wedge (y,y_\rho) \in \mathcal{R}_{\tau_2}\}$.
- For function types $\tau_1 \to \tau_2$, $\mathcal{R}_{\tau_1 \to \tau_2} := \{(f, f_\rho) \mid \forall (x,x_\rho) \in \mathcal{R}_{\tau_1}, (f(x), f_\rho(x_\rho)) \in \mathcal{R}_{\tau_2}\}$.

For an environment $\Gamma = (x_1 : \tau_1, \ldots, x_n : \tau_n)$, we define $\mathcal{R}_\Gamma$ to be $\mathcal{R}_{\tau_1 \times \cdots \times \tau_n}$.

We now show what is often called the *fundamental lemma* of a logical relations argument:

Lemma C.2. *For every term $\Gamma \vdash t : \tau$ in our language, and every environment $(\gamma, \gamma_\rho) \in \mathcal{R}_\Gamma$, $([\![\Gamma \vdash t : \tau]\!](\gamma), [\![\rho\{\Gamma\} \vdash \rho\{t\} : \rho\{\tau\}]\!](\gamma_\rho)) \in \mathcal{R}_\tau$.*

Proof sketch. We proceed by induction on the language's syntax:

- For constants $c : \tau$, we assume given constants $c_\rho : \rho\{\tau\}$ satisfying $(c, c_\rho) \in \mathcal{R}_\tau$. In particular, this means that primitive distributions, such as **flip** : $D\,\mathbb{B}$ must come equipped with densities, such as $\mathbf{flip}_\rho : \mathbb{B} \to \mathbb{R}$ (which would be $\lambda b.0.5$ in the case of a fair coin).
- For variables $x : \tau$, we appeal to the hypothesis that the environments (γ, γ_ρ) satisfy the relation.
- For $\mathbf{sample}(t_1, t_2)$, we first apply the inductive hypothesis to establish that $[\![\rho\{t_1\}]\!](\gamma_\rho)$ is a correct density function for the primitive distribution $[\![t_1]\!](\gamma)$. Then we can see that all three correctness criteria for translations of $M\,\tau$ terms are satisfied:
  1. $\pi_1([\![\mathbf{sample}(t_1,t_2)]\!](\gamma))$ is supported only on one possible trace shape, the shape $s = [([\![t_2]\!](\gamma), \sigma)]$, where σ is the type over which the primitive distribution $[\![t_1]\!](\gamma)$ is defined. Therefore, the density of a trace (s', v) is 0 if $s' \neq s$, and otherwise,

it is the density of v under $[\![t_1]\!](\gamma)$, with respect to $\mathbb{B}_\sigma$. Our translation outputs a density that is zero if $[\![t_2]\!](\gamma)$ does not appear in the input trace, and otherwise outputs the correct density of the value stored there with respect to $\mathbb{B}_\sigma$, using $\rho\{t_1\}$. If the trace contains *extra* entries, then the third return value from our translation will be a non-empty trace, and so we will still satisfy the requirement that $\lambda u.\underline{isempty}(\pi_3(p(u))) \cdot \pi_1(p(u))$ is a correct trace density (it will be zero when the third return value is non-empty).

2. As required, the second output of our translation applies the return-value function from the semantics of **sample** to produce the value returned by **sample** on a particular trace (the result of pop_σ).
3. By the definition of pop_σ, we have that when the given trace does contain the name $[\![t_2]\!](\gamma)$, the third return value of our translation will be the remainder of the trace.

- For **factor**, the third return value is the input trace, so the only trace to which we assign any mass is the empty trace (which accords with the semantics of **factor**). On that trace, our density is equal to $e^{[\![t]\!](\gamma)}$, which matches the semantics.
- For **do**$\{x \leftarrow t; m\}$, we appeal to the inductive hypothesis for the correctnes sof $\rho\{t\}$ and $\rho\{\mathbf{do}\{m\}\}$, and to the product rule for densities of joint distributions.
- The argument for $\lambda x.t$ and $t_1\ t_2$ (forming functions and applying them) is standard, and follows from the way we've defined $\mathcal{R}_{\tau_1 \to \tau_2}$. See, e.g., Huot et al. [6].

□

Having proven the fundamental lemma, the full correctness result is straight-forward.

Proposition C.3. *For* $\vdash p : M\,\tau$, $[\![\mathit{density}\{p\}]\!]$ *is a density of* $\pi_1([\![p]\!])$ *with respect to* $\mathbb{B}_{\mathit{Trace}}$.

Proof. By the fundamental lemma, $([\![p]\!], [\![\rho\{p\}]\!]) \in \mathcal{R}_{M\,\tau}$, which implies that $\lambda u.\underline{isempty}(\pi_3([\![\rho\{p\}]\!](u))) \cdot \pi_1([\![\rho\{p\}]\!](u))$ is a correct trace density of $\pi_1([\![p]\!])$. This function is precisely what $[\![\mathit{density}\{p\}]\!]$ computes. □

Weighted sampler. A similar argument applies to the weighted sampler program transformation, where our logical relations are now defined over $[\![\tau]\!] \times [\![\omega\{\tau\}]\!]$. These relations are:

- For ground types σ, $\mathcal{R}_\sigma = \{(x, x) \mid x \in [\![\sigma]\!]\}$.
- For distributions $D\,\sigma$, $\mathcal{R}_\sigma = \{(x, x) \mid x \in [\![D\,\sigma]\!]\}$, i.e., the transformation leaves primitive distributions unchanged.
- For products and functions, the same inductive definitions as in the logical relations for ρ.
- For probabilistic program types $M\,\tau$, $\mathcal{R}_{M\,\tau} = \{((\mu, f), (\nu, g)) \mid \nu \text{ a probability measure } \wedge (f, \pi_3 \circ g) \in \mathcal{R}_\tau \wedge \oiint w \odot \delta_u(\langle \pi_1, \pi_2 \rangle)_* \mathcal{V}(\nu, g)(d(u, w)) = \mu\}$. This says that the translation of a probabilistic program is another probabilistic program that is normalized (denotes a probability measure, not an unnormalized measure), has the same return-value-function over traces, and – when the returned weight is factored in – denotes the same measure over traces.

An analogous fundamental lemma can be proven for this logical relation, and then the correctness of *wsamp* follows. Together with the proposition above, this establishes Prop. 3.1.

D Sound Inference: Details

Unfolding the semantics of **do**, **let**, and **return**, we get that $\mathcal{V}[\![\mathit{importance}\{p, q\}]\!]$ is equal to

$$\oiint \delta_{(\hat{t}, \hat{w} \cdot \frac{[\![\mathit{density}\{p\}]\!](\hat{t})}{[\![\mathit{density}\{q\}]\!](\hat{t})})} \mathcal{V}[\![\mathit{wsamp}\{q\}]\!](d(\hat{t}, \hat{w})).$$

Using the correctness proof for densities, this can be rewritten to

$$\oiint \delta_{(\hat{t}, \hat{w} \cdot \frac{d(\pi_1 \circ [\![p]\!])}{d(\pi_1 \circ [\![q]\!])}(\hat{t}))} \mathcal{V}[\![\mathit{wsamp}\{q\}]\!](d(\hat{t}, \hat{w})).$$

The pushforward of this measure by π_1 simply returns the sampled $\hat{t}$ unchanged, and so is clearly equal to the pushforward of $\mathcal{V}[\![\mathit{wsamp}\{q\}]\!]$ by π_1. Furthermore, letting $\nu = \mathcal{V}[\![\mathit{importance}\{p, q\}]\!]$, we have

$$\begin{aligned}
\oiint \pi_2(x) \odot \delta_{\pi_1(x)} \nu(dx) &= \oiint (\hat{w} \cdot \frac{d(\pi_1 \circ [\![p]\!])}{d(\pi_1 \circ [\![q]\!])}(\hat{t}) \odot \delta_{\hat{t}} \mathcal{V}[\![\mathit{wsamp}\{q\}]\!](d(\hat{t}, \hat{w})) \\
&= \oint \frac{d(\pi_1 \circ [\![p]\!])}{d(\pi_1 \circ [\![q]\!])}(\hat{t}) \odot \delta_{\hat{t}}(\pi_1 \circ [\![q]\!])(d\hat{t}) \\
&= \oint \delta_{\hat{t}}(\pi_1 \circ [\![p]\!])(d\hat{t}) \\
&= \pi_1 \circ [\![p]\!],
\end{aligned}$$

as desired. The second line uses the correctness result for *wsamp*, and the third uses the definition of the Radon-Nikodym derivative.

E The Traced PPL Landscape

One thesis of this abstract is that a useful lens through which to view the landscape of existing PPLs is to ask the following questions:

- What core operations, like *density* and *wsamp*, do the languages implement?
- How does the choice of core operations affect what constructs the language can expose?
- How does the choice of core operations affect the inference algorithms that can be automated using those core operations?

A full study of these questions for different languages is left to future work, but we make some observations here, pointing to directions that may be interesting to explore further:

- In the Gen PPL [3], there is no **factor** statement, and all programs encode *probability* measures over traces. Instead of *wsamp*, Gen programs support exact (unweighted) simulation.
- In ProbTorch [17], there is a core operation that partially constrains the execution of a probabilistic program using the trace of another program, but (unlike our *density* operation) allows the trace to be incomplete or to contain extraneous variables not sampled by the program. It is still unclear to us how exactly to state the general specification and correctness theorem for this operation, in terms of the measure over traces denoted by the program. Gen features a similar but more restricted operation, called `generate`, which, given a partial trace as input, returns a properly weighted sample for the *posterior* over complete traces, conditioning on the partial trace. (This is more restricted than ProbTorch, in that Gen does not permit the partial trace to contain extraneous or auxiliary variables that are *not* present in the model.)
- The Pyro [1] and Gen [3] languages contain control flow combinators, like `plate` in Pyro or `Map` in Gen, that are semantically equivalent to particular loops or recursions, but are translated specially by the PPL's automated operations, to yield more efficient implementations of densities, gradient estimators, or other operations.