Probabilistic Programming with Programmable Variational Inference

Probabilistic Programming with Programmable Variational Inference

MCCOY R. BECKER*, MIT, USA
ALEXANDER K. LEW*, MIT, USA
XIAOYAN WANG, MIT, USA
MATIN GHAVAMI, MIT, USA
MATHIEU HUOT, MIT, USA
MARTIN C. RINARD, MIT, USA
VIKASH K. MANSINGHKA, MIT, USA

Compared to the wide array of advanced Monte Carlo methods supported by modern probabilistic programming languages (PPLs), PPL support for *variational inference* (VI) is less developed: users are typically limited to a predefined selection of variational objectives and gradient estimators, which are implemented monolithically (and without formal correctness arguments) in PPL backends. In this paper, we propose a more modular approach to supporting variational inference in PPLs, based on compositional program transformation. In our approach, variational objectives are expressed as programs, that may employ first-class constructs for computing *densities of* and *expected values under* user-defined models and variational families. We then transform these programs systematically into unbiased gradient estimators for optimizing the objectives they define. Our design enables modular reasoning about many interacting concerns, including automatic differentiation, density accumulation, tracing, and the application of unbiased gradient estimation strategies. Additionally, relative to existing support for VI in PPLs, our design increases expressiveness along three axes: (1) it supports an open-ended set of user-defined variational objectives, rather than a fixed menu of options; (2) it supports a combinatorial space of gradient estimation strategies, many not automated by today's PPLs; and (3) it supports a broader class of models and variational families, because it supports constructs for approximate marginalization and normalization (previously introduced only for Monte Carlo inference). We implement our approach in an extension to the Gen probabilistic programming system (`genjax.vi`, implemented in JAX), and evaluate our automation on several deep generative modeling tasks, showing minimal performance overhead vs. hand-coded implementations and performance competitive with well-established open-source PPLs.

CCS Concepts: • **Software and its engineering → Semantics**; • **Mathematics of computing → Variational methods**; **Bayesian computation**; *Statistical software*.

Additional Key Words and Phrases: probabilistic programming, automatic differentiation, variational inference

ACM Reference Format:
McCoy R. Becker, Alexander K. Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C. Rinard, and Vikash K. Mansinghka. 2024. Probabilistic Programming with Programmable Variational Inference. *Proc. ACM Program. Lang.* 8, PLDI, Article 233 (June 2024), 25 pages. https://doi.org/10.1145/3656463

*Equal contribution.

Authors' addresses: McCoy R. Becker, MIT, Cambridge, USA, mccoyb@mit.edu; Alexander K. Lew, MIT, Cambridge, USA, alexlew@mit.edu; Xiaoyan Wang, MIT, Cambridge, USA, xyw@mit.edu; Matin Ghavami, MIT, Cambridge, USA, mghavami@mit.edu; Mathieu Huot, MIT, Cambridge, USA, mhuot@mit.edu; Martin C. Rinard, MIT, Cambridge, USA, rinard@mit.edu; Vikash K. Mansinghka, MIT, Cambridge, USA, vkm@mit.edu.

ACM 2475-1421/2024/6-ART233
https://doi.org/10.1145/3656463

1 INTRODUCTION

Variational inference (VI) is a popular approach to two fundamental probabilistic modeling tasks:

- **Fitting probabilistic models to data.** Given a family of joint probability distributions $\mathcal{P} = \{P_\theta(x, y) \mid \theta \in \mathbb{R}^n\}$ defined over *latent variables* x and *observed variables* y, find the one that best explains an observed dataset $\mathbf{y}$. For example, writing p_θ for the probability density function of P_θ, we may be interested in finding $\theta \in \mathbb{R}^n$ that maximizes the *marginal likelihood*

$$p_\theta(\mathbf{y}) = \int_X p_\theta(x, \mathbf{y}) dx. \tag{1}$$

- **Approximating intractable posterior distributions.** For a particular probabilistic model $P_\theta(x, y)$, find the best approximation to the (usually intractable) posterior distribution $P_\theta(x \mid \mathbf{y})$, from a class $\mathcal{Q} = \{Q_\phi(x) \mid \phi \in \mathbb{R}^m\}$ of tractable approximations (the *variational family*). For example, again using lower-case letters for probability density functions, we may be interested in finding ϕ that minimizes the *reverse KL divergence*

$$D_{KL}(Q_\phi(x) \,||\, P_\theta(x \mid \mathbf{y})) = -\mathbb{E}_{x \sim Q_\phi}\left[\log \frac{p_\theta(x \mid \mathbf{y})}{q_\phi(x)}\right]. \tag{2}$$

Practitioners often aim to solve both these tasks at once, simultaneously fitting a probabilistic model and approximating its posterior distribution. To do so, one defines a *variational objective* $\mathcal{F} : \mathcal{P} \times \mathcal{Q} \to \mathbb{R}$, mapping particular distributions P_θ and Q_ϕ to a scalar *loss* (or *reward*). For example, one common choice is the *evidence lower bound*, or ELBO:

$$\mathbf{ELBO}(P, Q) := \mathbb{E}_{x \sim Q}[\log p(x, \mathbf{y}) - \log q(x)] = \log p(\mathbf{y}) - D_{KL}(Q(x) || P(x \mid \mathbf{y})) \tag{3}$$

As the decomposition on the right-hand side suggests, maximizing the ELBO simultaneously maximizes the (log) marginal likelihood of the data $\mathbf{y}$ and minimizes the KL divergence of the posterior approximation Q to the posterior. Besides the ELBO, researchers have also proposed many alternative objectives [1, 10, 11, 15, 24, 46, 59, 60, 65], which formalize the two goals of *fitting models* and *approximating posteriors* differently (e.g., by using divergences other than the KL). Once a variational objective has been defined, practitioners aim to find parameters (θ, ϕ) that maximize (or minimize) $\mathcal{F}(P_\theta, Q_\phi)$. There are many possible approaches to performing this optimization. The most popular methods rely on *gradients* $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$ of the objective—or more often, unbiased stochastic estimates of these gradients. Designing and implementing algorithms for estimating these gradients, with sufficiently low variance and computational expense, is the key roadblock on the path from *defining* a variational inference problem to solving it.

Indeed, although variational inference algorithms have found widespread adoption in Bayesian statistics [7, 8, 19, 25, 32] and in probabilistic deep learning [27, 29, 46, 57, 70], implementing variational inference algorithms by hand remains a tedious and error-prone endeavor. The key mathematical ingredients specifying a variational inference problem—$\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{F}$—are typically not represented directly in code; rather, the practitioner must:

(1) use algebra, probability theory, and calculus to derive a *gradient estimator*: a way to rewrite the gradient $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$ as an expectation $\mathbb{E}_{y \sim M_{(\theta,\phi)}}[f_{(\theta,\phi)}(y)]$, for some family of distributions M and some family of functions f; and then
(2) write code to sample $y \sim M_{(\theta,\phi)}$ and evaluate $f_{(\theta,\phi)}(y)$—an unbiased estimate of $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$.

It is often non-trivial to ensure that M and f are faithfully implemented, and that the math used to derive them in the first place is error-free. Small changes to $\mathcal{F}$, $\mathcal{P}$, or $\mathcal{Q}$, or to the gradient estimation strategy employed in step (1), can require large, non-local changes to M and f, and in implementing these changes, it is easy to introduce hard-to-detect bugs. When optimization fails, it is often unclear whether the problem is with the math, the code, or just the hyperparameters.

Automation via Probabilistic Programming. Reflecting the importance of variational inference, many probabilistic programming languages (PPLs) (especially "deep" PPLs, such as Pyro [6], Edward [68, 69], and ProbTorch [67]), feature varying degrees of automation for VI workflows. In these languages, users can express both models $\mathcal{P}$ and variational families $\mathcal{Q}$ as probabilistic programs; the system then automates the estimation of gradients for a pre-defined set of supported variational objectives $\mathcal{F}$. This design significantly lowers the cost of implementing and iterating on variational inference algorithms, but several pain points remain:

- **Incomplete coverage.** Existing PPLs offer limited or no support for many variational objectives, including forward KL objectives [52]; hierarchical, nested, or recursive variational objectives [37, 60, 74]; symmetric divergences [16]; trajectory-balance objectives [46]; SMC-based objectives [22, 42, 45, 51]; and others. Today's PPLs also do not automate many powerful gradient estimation strategies, for example those based on measure-valued differentiation [50].
- **Duplicative engineering effort.** For PPL maintainers, supporting new gradient estimation strategies or language features requires separately introducing the same logic into the implementations of multiple variational objectives. Because this engineering effort is non-trivial, many capabilities are not uniformly supported. For example, as of this writing, Pyro's `ReweightedWakeSleep` objective [33] does not support minibatching, even though other objectives do. As another example, variance reduction strategies such as data-dependent baselines and enumeration of discrete latents are implemented for the ELBO in Pyro, but not, e.g., for the importance-weighted ELBO.
- **Difficulty of reasoning.** The monolithic implementations of each variational objective's gradient estimation logic intertwine various concerns, including log density accumulation, automatic differentiation, gradient propagation through stochastic choices, and variance reduction logic. This can make it difficult to reason about correctness. Indeed, while the community has made tremendous progress in understanding the compositional correctness arguments of an increasingly broad class of Monte Carlo inference methods for probabilistic programs [9, 39, 43, 75, 76], pioneering work on correctness for variational inference [34, 35, 41] has generally focused on specific properties (e.g., smoothness and absolute continuity) in somewhat restricted languages, and not to end-to-end correctness of gradient estimation for variational inference.

This Work. In this paper, we present a highly modular, programmable approach to supporting variational inference in PPLs. In our approach, all three ingredients of the variational inference problem—$\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{F}$—are encoded as programs in expressive probabilistic languages, which support compositional annotation for specifying the desired mix of gradient estimation strategies. We then use a sequence of modular program transformations—each of which we independently prove correct—to construct unbiased gradient estimators for the user's variational objective.

Contributions. This paper contributes:

- **Languages for models, variational families, and variational objectives**: We present an expressive language for models and variational families (§3.2), similar to Gen, ProbTorch, or Pyro, along with an expressive differentiable language for variational objective functions (§3.3).
- **Flexible, modular automation**: We automate a broad class of unbiased gradient estimators for variational objectives (§5). New primitive gradient estimation strategies can be added modularly with just a few lines of code, without deeply understanding system internals (Appx. F).
- **Formalization**: We formalize our approach as a sequence of composable program transformations (§4-5) of simply-typed λ-calculi for probabilistic programs (§3), and prove the unbiasedness of gradient estimation (under mild technical conditions) by logical relations (§6). Ours is the first formal account of variational inference for PPLs that accounts for the interactions between tracing, density computation, gradient estimation strategies, and automatic differentiation.

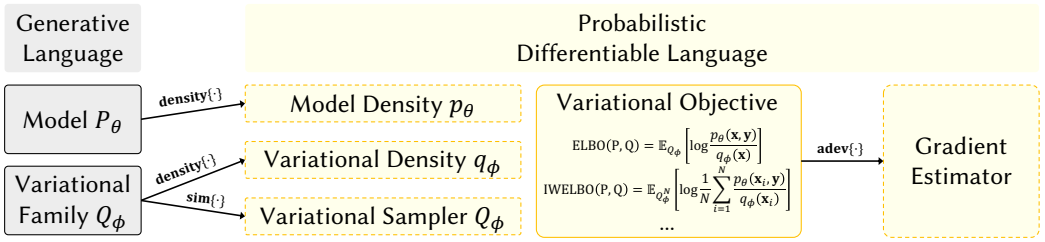

Fig. 1. We compose multiple program transformations to automate the construction of unbiased gradient estimators for variational inference. The user begins by writing programs in the generative language (gray) to encode a model and variational family. These programs are compiled into procedures for density evaluation and simulation in the differentiable language (yellow). These automated procedures can be used to concisely define a variational objective. We can then apply the ADEV differentiation algorithm to automatically construct a *gradient estimator*, which unbiasedly estimates gradients of the variational objective. Solid outlines indicate user-written programs, whereas dashed outlines indicate automatically constructed programs.

- **System**: We contribute `genjax.vi`, a performant, GPU-accelerated implementation of our approach in JAX [20], which also extends our formal modeling language with constructs for marginalization and normalization (§7) [39]. We also contribute concise, pedagogical Haskell and Julia versions. Our implementations are the first to feature *reverse-mode* variants of the ADEV algorithm for modularly differentiating higher-order probabilistic programs [40].[1]
- **Empirical evaluation**: We evaluate `genjax.vi` on several benchmark tasks, including the challenging Attend-Infer-Repeat model [17]. We find that `genjax.vi` makes it possible to encode new gradient estimators that converge faster and to better solutions than Pyro's estimators. We also show, for the first time, that a version of ADEV can be scalably and performantly implemented, to deliver competitive performance on realistic probabilistic deep learning workloads.[2]

Programmability is sometimes seen as being at odds with automation. We emphasize that this paper *expands* the automation provided by the system, relative to existing VI support in PPLs, by automating a combinatorial space of gradient estimators for arbitrary objectives specified as programs (rather than the handful of objectives and estimators supported by existing PPLs).

2 OVERVIEW

Fig. 1 illustrates the workflow of a typical user of our system for modular variational inference:

- **Model and variational programs.** The user begins by writing two probabilistic programs in our *generative language*: a *model program* and a *variational program*. The generative language is a trace-based PPL that resembles Pyro [6], ProbTorch [67], and Gen [14]. The model program encodes a family of joint probability distributions $P_\theta(x, y)$, and the variational program encodes a family of distributions $Q_\phi(x)$, possible approximations to the posterior $P_\theta(x \mid \mathbf{y})$ for data $\mathbf{y}$.
- **Objective function.** The user now seeks to find values of (θ, ϕ) that simultaneously (1) fit P_θ to the data $\mathbf{y}$, and (2) make Q_ϕ close to the posterior $P_\theta(x \mid \mathbf{y})$. To make these informal desiderata precise, the user defines an *objective function*, using our *differentiable language*. This language features constructs for taking expectations with respect to, and evaluating densities of, generative language programs, making it easy to concisely express objectives like the ELBO (Eqn. 3).
- **Gradient estimator.** The final step is to optimize the objective function via stochastic gradient ascent. We construct unbiased gradient estimators for the user's objective function with an

[1] System and code available at https://gen.dev/genjax/vi
[2] Lew et al. [40] present only a toy Haskell implementation of forward-mode ADEV, and report no experiments.

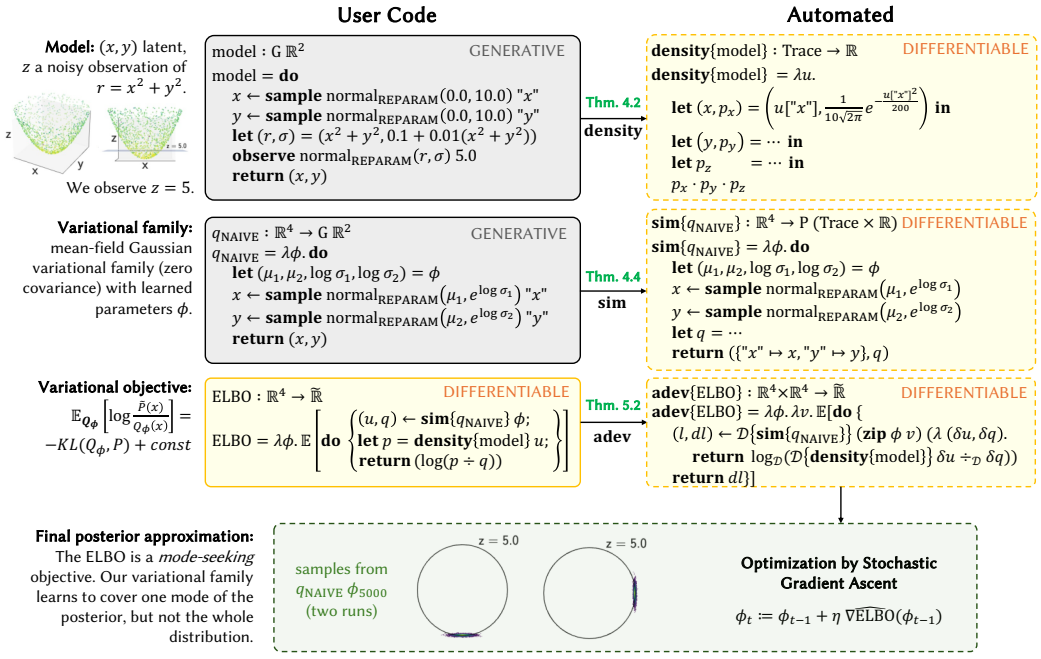

Fig. 2. An illustration of our modular approach to automating variational inference, on a toy example. **(Top)** Users write *generative* code to define a *model* and a *variational family*. Automated program transformations, formalized and proven correct in §4, compile *differentiable* code for evaluating densities and simulating traces. **(Middle)** Users write a program in the differentiable language to define a variational objective, in this case the *evidence lower bound* (ELBO). This code may invoke compiled simulators and density evaluators for generative programs. The **adev** transformation automates an unbiased gradient estimator for the objective. **(Bottom)** The gradients are used for optimization, training the variational family to approximate the posterior.

extended version of the ADEV algorithm [40]. Users can rapidly explore a combinatorial space of estimation strategies with compositional annotations on their generative programs, to navigate tradeoffs between the variance and the computational cost of the automated gradient estimator.

Example. To make this concrete, consider the toy problem illustrated in Fig. 2, which we seek to solve by training a variational approximation to the posterior. We go through the following steps:

- **Define a model**. Our model encodes a generative process for points (x, y, z) around a 3D cone. We use **sample** to sample latents x and y with string-valued *names*, and **observe** to condition on the observation that $z = 5.0$. Our goal is to infer (x, y) consistent with this observation.
- **Define a variational family**. In the second panel of Fig. 2, we construct a *variational family*, a parametric family of possible approximations to the posterior distribution. Our variational inference task will be to learn parameters that maximize the quality of the approximation. Our `qNAIVE` is a *mean-field* variational approximation, i.e., it generates x and y independently. Note that primitive distributions (here, `normal`) are annotated with gradient estimation strategies (here, `REPARAM`) for propagating derivative information through the corresponding primitive.[3]

[3]For a primitive distribution μ_θ over values of type X, parameterized by arguments $\theta \in \mathbb{R}^n$, a gradient estimation strategy for μ_θ is an approach to unbiasedly estimating $\nabla_\theta \mathbb{E}_{x\sim\mu_\theta}[f(\theta, x)]$ for functions $f : \mathbb{R}^n \times X \to \mathbb{R}$. ADEV composes these primitive estimation strategies into composite strategies for estimating gradients of variational objectives.

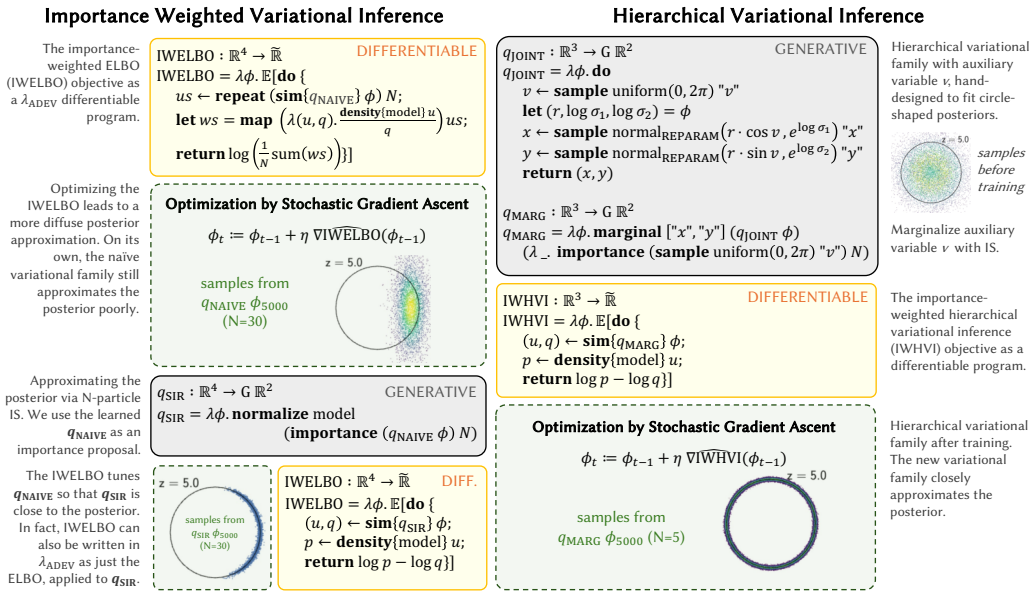

Fig. 3. With programmable VI, users can define their own variational objectives, and use new modeling language features to program more expressive models and variational families. Here, we apply importance-weighted VI [11] and hierarchical VI [60, 65] to the toy problem from Fig. 2.

- **Define a variational objective**. We now use the differentiable language to define the objective function we wish to optimize. Three constructs are especially useful: (1) **density**, which computes or estimates densities of probabilistic programs; (2) **sim**, which generates a pair (x, w) of a sample and its density from a probabilistic program; and (3) $\mathbb{E}$, which takes the expected value of a stochastic procedure. We use them together to implement the ELBO objective from Eqn. 3.
- **Perform stochastic optimization.** Our objective is compiled into an *unbiased estimator* of its *gradient* with respect to the input parameters ϕ. We can then apply stochastic optimization algorithms, such as stochastic gradient ascent and ADAM. The bottom of Fig. 2 illustrates samples from q_{NAIVE} after training. Because the ELBO minimizes the *reverse* (or *mode-seeking*) KL divergence, our variational approximation learns to hug one edge of the circle-shaped posterior.

Better Inference with Programmable Objectives. To better fit the whole posterior, we will need either a new objective function or a more expressive variational family. Our system's programmability allows users to quickly iterate within a broad space of VI algorithms. In Fig. 3, we illustrate two possible strategies for improving the posterior approximation:

- **Importance-weighted variational inference.** On the left side of Fig. 3, we define the `IWELBO` variational objective [11], as the expected value of the process that generates N *particles* from the variational family q_{NAIVE}, and computes a log mean importance weight. This objective does not directly encourage q_{NAIVE} to approximate the posterior well; rather, it encourages q_{NAIVE} to be a good *proposal distribution* for N-particle importance sampling, targeting the posterior. To illustrate this, we define a new probabilistic program q_{SIR}, which uses **normalize** to construct a *sampling importance resampling* (SIR) approximation to the posterior: it generates N samples

Supported strategies vary by primitive; for the Normal distribution, for instance, they include `REPARAM`, `MEASURE-VALUED`, and `REINFORCE`, corresponding to different approaches to gradient estimation from the literature. See §5.

from $\mathsf{q}_{\texttt{NAIVE}}$, computes *importance weights* for each sample, and randomly selects a sample to return according to the weights. Drawing samples from $\mathsf{q}_{\texttt{SIR}}$, with $N = 30$ and ϕ set to the parameters that optimized the IWELBO objective, yields a close approximation to a larger portion of the posterior. (In fact, the IWELBO objective can be equivalently expressed in our framework as the ordinary ELBO, but applied directly to $\mathsf{q}_{\texttt{SIR}}$ rather than to $\mathsf{q}_{\texttt{NAIVE}}$.)

- **Hierarchical variational families.** On the right side of Fig. 3, we define a better variational family: $\mathsf{q}_{\texttt{MARG}}$. First, we define a hand-designed posterior approximation $\mathsf{q}_{\texttt{JOINT}}$, which extends $\mathsf{q}_{\texttt{NAIVE}}$ with an *auxiliary variable* v, to allow it to better fit circle-shaped posteriors. We cannot directly use $\mathsf{q}_{\texttt{JOINT}}$ with the ELBO objective, however, because it is defined over the space of triples (v, x, y), not the space of pairs (x, y) as in our model. The **marginal** construct shrinks the sample space back down to just (x, y), approximating the marginal density $\mathsf{q}_{\texttt{MARG}}(x, y)$ using importance sampling. The resulting algorithm is an instance of importance weighted HVI (IWHVI) [65].

3 SYNTAX AND SEMANTICS

We now formalize the core of our approach. Although our formal model lacks several features of our full language (see §7 and Appx. A), it is still expressive, featuring higher-order functions, continuous and discrete sampling, stochastic control flow, and discontinuous branches. Ultimately, our formalization aims to give an account of how user programs representing models, variational families, and variational objectives are transformed into compiled programs representing unbiased gradient estimators. We begin in this section by introducing two calculi for generative and differentiable probabilistic programming (Fig. 4).

3.1 Shared Core

3.1.1 Syntax. The top of Fig. 4 presents the *shared core*, the λ-calculus on which both our languages build. It is largely standard, with functions, tuples, if statements, and ground types for Booleans, strings, and numbers, but two aspects merit further discussion:

- *Smooth and non-smooth reals.* First, following Lew et al. [40], we have two types for real numbers, $\mathbb{R}$ and $\mathbb{R}^*$. Intuitively, the type $\mathbb{R}$ is the type of "real numbers that must be used differentiably," whereas the type $\mathbb{R}^*$ is the type of "real numbers that may be manipulated in any (measurable) way." These constraints are enforced by the types of primitive functions: non-smooth primitives like $< : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{B}$ only accept $\mathbb{R}^*$ inputs. Smooth primitives, by contrast, come in two varieties, smooth versions (e.g. $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$) and non-smooth versions (e.g. $+^* : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{R}^*$).[4] We allow implicit promotion of terms of type $\mathbb{R}^*$ into terms of type $\mathbb{R}$.
- *Primitive distributions.* The shared core includes a type $D\ \sigma$ of primitive probability distributions over ground types σ. Again following Lew et al. [40], we expose multiple versions of each primitive, e.g. $\mathbf{normal}_{\text{REPARAM}}$ and $\mathbf{normal}_{\text{REINFORCE}}$. All versions denote the same distribution, and so our correctness results (which are phrased in terms of our denotational semantics) ensure that gradients will target the same objective no matter which version a program uses (Thm. 5.2). But different versions of the same primitive employ different *estimation strategies* for propagating derivatives, striking different trade-offs between variance and cost. Furthermore, because different estimation strategies may place different requirements on the user's program, the typing rules for different versions of the same primitive may differ. For example, $\mathbf{normal}_{\text{REINFORCE}}$ constructs a distribution of type $D\ \mathbb{R}^*$, meaning that probabilistic programs that draw samples from it can

[4]The reader may wonder if we also need primitives that accept some smooth and some non-smooth inputs, but this turns out to be unnecessary, because non-smooth inputs can always be safely promoted to smooth inputs. The output will then also be smooth, but this is by design: if *any* input to a primitive has smooth type, then the primitive's output must also have smooth type, to ensure that future computation does not introduce non-differentiability.

Shared Core

Ground types $\sigma ::= 1 \mid \mathbb{B} \mid \mathbb{I} \mid \mathbb{R}_{>0} \mid \mathbb{R}_{\geq 0} \mid \mathbb{R} \mid \mathbb{R}^* \mid \mathrm{Str} \mid \sigma_1 \times \sigma_2$ Types $\tau ::= \sigma \mid D\,\sigma \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$

Terms $t ::= () \mid r \mid c \mid x \mid \lambda x.t \mid t_1\ t_2 \mid (t_1, t_2) \mid \pi_1\ t \mid \pi_2\ t \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$

Primitives $c ::= + \mid +^* \mid < \mid \mathbf{normal}_{\mathrm{REPARAM}} \mid \mathbf{normal}_{\mathrm{REINFORCE}} \mid \mathbf{flip}_{\mathrm{REINFORCE}} \mid \mathbf{flip}_{\mathrm{ENUM}} \mid \mathbf{flip}_{\mathrm{MVD}} \mid \ldots$

$$\frac{t : \mathbb{R}^*}{t : \mathbb{R}} \qquad \frac{}{+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}} \qquad \frac{}{+^* : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{R}^*} \qquad \frac{}{< : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{B}}$$

$$\frac{}{\mathbf{normal}_{\mathrm{REPARAM}} : \mathbb{R} \times \mathbb{R}_{>0} \to D\,\mathbb{R}} \qquad \frac{}{\mathbf{normal}_{\mathrm{REINFORCE}} : \mathbb{R} \times \mathbb{R}_{>0} \to D\,\mathbb{R}^*} \qquad \frac{}{\mathbf{flip}_{\mathrm{MVD}} : \mathbb{I} \to D\,\mathbb{B}}$$

$$[\![1]\!] = \{()\} \quad [\![\mathbb{I}]\!] = [0,1] \quad [\![\mathbb{R}]\!] = [\![\mathbb{R}^*]\!] = \mathbb{R} \quad [\![D\,\sigma]\!] = \mathrm{Prob}_{\ll \mathcal{B}_\sigma}\,[\![\sigma]\!]$$
$$[\![\mathbf{normal}_{\mathrm{REPARAM}}]\!] = [\![\mathbf{normal}_{\mathrm{REINFORCE}}]\!] = \lambda(\mu,\sigma).\mathcal{N}(\mu,\sigma)$$

Generative Probabilistic Programming (λ_{Gen})

Types $\tau ::= G\,\tau$ (generative programs)

Terms $t ::= \mathbf{return}\ t \mid \mathbf{sample}\ t_1\ t_2 \mid \mathbf{observe}\ t_1\ t_2 \mid \mathbf{do}\{m\}$

$m ::= t \mid x \leftarrow t; m$

$$\frac{t : \tau}{\mathbf{return}\ t : G\,\tau} \qquad \frac{t_1 : D\,\sigma \quad t_2 : \mathrm{Str}}{\mathbf{sample}\ t_1\ t_2 : G\,\sigma}$$
$$\frac{t_1 : D\,\sigma \quad t_2 : \sigma}{\mathbf{observe}\ t_1\ t_2 : G\,1} \qquad \frac{t : G\,\tau}{\mathbf{do}\{t\} : G\,\tau}$$
$$\frac{t : G\,\tau_1 \quad x : \tau_1 \vdash \mathbf{do}\,\{m\} : G\,\tau_2}{\mathbf{do}\{x \leftarrow t; m\} : G\,\tau_2}$$

$$[\![G\,\tau]\!] = \mathrm{Meas}^{DS}_{\ll \mathcal{B}_{\mathbb{T}}}\ \mathbb{T} \times (\mathbb{T} \Rightarrow [\![\tau]\!])$$
$$[\![\mathbf{return}\ t]\!]_1(\gamma, U) = \delta_{\{\}}(U)$$
$$[\![\mathbf{return}\ t]\!]_2(\gamma, u) = [\![t]\!](\gamma)$$
$$[\![\mathbf{sample}\ t_1\ t_2]\!]_1(\gamma, U) = \int [\![t_1]\!](\gamma, dv)\,\delta_{\{[\![t_2]\!](\gamma) \mapsto v\}}(U)$$
$$[\![\mathbf{sample}\ t_1\ t_2]\!]_2(\gamma, u) = u[[\![t_2]\!](\gamma)]$$
$$[\![\mathbf{observe}\ t_1\ t_2]\!]_1(\gamma, U) = \frac{d([\![t_1]\!](\gamma))}{d\mathcal{B}_\sigma}([\![t_2]\!](\gamma))\,\delta_{\{\}}(U)$$
$$[\![\mathbf{observe}\ t_1\ t_2]\!]_2(\gamma, u) = ()$$
$$[\![\mathbf{do}\{x \leftarrow t; m\}]\!]_1(\gamma, U) = \int [\![t]\!]_1(\gamma, du_1) \int [\![\mathbf{do}\{m\}]\!]_1(\gamma', du_2)\ \delta_{u_1 + u_2}(U) \cdot \mathit{disj}(u_1, u_2)$$
$$\text{where } \gamma' = \gamma[x \mapsto [\![t]\!]_2(\gamma, u_1)]$$
$$[\![\mathbf{do}\{x \leftarrow t; m\}]\!]_2(\gamma, u) = [\![\mathbf{do}\{m\}]\!]_2(\gamma', u')$$
$$\text{where } \gamma' = \gamma[x \mapsto [\![t]\!]_2(\gamma, u)]$$
$$\text{and } u' = \pi_2(\mathit{split}_{[\![t]\!]_1(\gamma)}(u))$$

Differentiable Probabilistic Programming (λ_{ADEV})

Types $\tau ::= P\,\tau \mid \widetilde{\mathbb{R}} \mid \mathrm{Trace}$

Terms $t ::= \mathbb{E}\,t \mid \mathbf{return}\ t \mid \mathbf{sample}\ t \mid \mathbf{do}\{m\} \mid \mathbf{score}\ t \mid \{\} \mid \{t_1 \mapsto t_2\} \mid t_1 + t_2 \mid t_1[t_2]$

$m ::= t \mid x \leftarrow t; m$

$$\frac{t : \tau}{\mathbf{return}\ t : P\,\tau} \qquad \frac{t : D\,\sigma}{\mathbf{sample}\ t : P\,\sigma}$$
$$\frac{t : \mathbb{R}_{\geq 0}}{\mathbf{score}\ t : P\,1} \qquad \frac{t : P\,\tau}{\mathbf{do}\{t\} : P\,\tau} \qquad \frac{t : P\,\mathbb{R}}{\mathbb{E}\,t : \widetilde{\mathbb{R}}}$$
$$\frac{t : P\,\tau_1 \quad x : \tau_1 \vdash \mathbf{do}\,\{m\} : P\,\tau_2}{\mathbf{do}\{x \leftarrow t; m\} : P\,\tau_2}$$

$$[\![P\,\tau]\!] = \mathrm{Meas}\,[\![\tau]\!] \quad | \quad [\![\widetilde{\mathbb{R}}]\!] = \mathrm{Meas}\,\mathbb{R} \quad | \quad [\![\mathrm{Trace}]\!] = \mathbb{T}$$
$$[\![\mathbf{return}\ t]\!](\gamma, U) = \delta_{[\![t]\!](\gamma)}(U)$$
$$[\![\mathbf{sample}\ t]\!](\gamma, U) = [\![t]\!](\gamma, U)$$
$$[\![\mathbf{score}\ t]\!](\gamma, U) = [\![t]\!](\gamma) \cdot \delta_{()}(U)$$
$$[\![\mathbf{do}\{x \leftarrow t; m\}]\!](\gamma, U) = \int [\![t]\!](\gamma, du_1)\ [\![\mathbf{do}\{m\}]\!](\gamma[x \mapsto u_1], U)$$
$$[\![\mathbb{E}\,t]\!](\gamma, U) = [\![t]\!](\gamma, U)$$
$$[\![\{\}]\!](\gamma) = \{\}$$
$$[\![\{t_1 \mapsto t_2\}]\!](\gamma) = \{[\![t_1]\!](\gamma) \mapsto [\![t_2]\!](\gamma)\}$$
$$[\![t_1 + t_2]\!](\gamma) = \begin{cases} u_1 + u_2 & \text{if } \mathit{disj}(u_1, u_2) \\ \{\} & \text{otherwise} \end{cases}$$
$$\text{where } u_i = [\![t_i]\!](\gamma)$$
$$[\![t_1[t_2]]\!](\gamma) = \begin{cases} u[v] & \text{if } v \in u \\ \mathrm{default}_\sigma & \text{otherwise} \end{cases}$$
$$\text{where } (u, v) = ([\![t_1]\!](\gamma), [\![t_2]\!](\gamma))$$

Fig. 4. Grammars, selected typing rules, and selected denotations for our core languages λ_{Gen} and λ_{ADEV}.

freely manipulate those samples. By contrast, $\mathbf{normal}_{\mathrm{REPARAM}}$ constructs a distribution of type $D\,\mathbb{R}$, so samples from $\mathbf{normal}_{\mathrm{REPARAM}}$ must be used smoothly.

3.1.2 Denotational Semantics. We assign to each type τ a mathematical space $[\![\tau]\!]$, and interpret an open term $\Gamma \vdash t : \tau$ as a map from $[\![\Gamma]\!] \coloneqq \prod_{x \in \Gamma} [\![\Gamma(x)]\!]$, the space of *environments* for context Γ, to $[\![\tau]\!]$, the space of results to which t can evaluate. Formally, we work in the category of *quasi-Borel spaces* [23], but to ease exposition, we present our semantics in terms of standard measure theory when possible. So, for example, we write that $[\![\mathbb{R}]\!]$ is the measurable space $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$, that $[\![\sigma_1 \times \sigma_2]\!]$ is the product of the measurable spaces $[\![\sigma_1]\!]$ and $[\![\sigma_2]\!]$, and so on, but with the implicit understanding that these can equivalently be viewed as quasi-Borel spaces. For semantics of higher-order types, we use quasi-Borel spaces explicitly (e.g., we set $[\![\tau_1 \to \tau_2]\!]$ to $[\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$, the quasi-Borel space of quasi-Borel maps from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$).

To give a semantics to our primitive distributions, we need to first assign to each ground type σ a *base measure* $\mathcal{B}_\sigma$ over $[\![\sigma]\!]$: for discrete types $\sigma \in \{1, \mathbb{B}, \mathrm{Str}\}$, we set $\mathcal{B}_\sigma(U) = |U|$, the counting measure, and for continuous types $\sigma \in \{\mathbb{R}, \mathbb{R}^*\}$, we set $\mathcal{B}_\sigma(U) = \int_{\mathbb{R}} 1_U(u)du$, the Lebesgue measure. Given two ground types σ_1 and σ_2, the base measure of the product, $\mathcal{B}_{\sigma_1 \times \sigma_2}$, is $\mathcal{B}_{\sigma_1} \otimes \mathcal{B}_{\sigma_2}$, the product of the base measures for each type. With base measures in hand, we can define $[\![D\ \sigma]\!] \coloneqq \mathrm{Prob}_{\ll \mathcal{B}_\sigma}[\![\sigma]\!]$, the space of probability measures on $[\![\sigma]\!]$ that are *absolutely continuous* with respect to $\mathcal{B}_\sigma$. Absolute continuity ensures that these distributions have *density functions*.

3.2 Generative Probabilistic Programming with $\lambda_{\mathbf{Gen}}$

Users write probabilistic models and variational families in λ_{Gen}, which extends the shared core with a monadic type $G\ \tau$ of *generative programs* (Fig. 4, left). Examples of programs in λ_{Gen} include the model and variational programs (`model` and $\mathsf{q}_{\mathtt{NAIVE}}$) in Fig. 2.

3.2.1 Syntax. Syntactically, programs of type $G\ \tau$ interleave standard functional programming logic (from the shared core) with two new kinds of statements: **sample** and **observe**. The **sample** statement takes as input a probability distribution to sample (of type $D\ \sigma$) and a unique *name* for the random variable being sampled (of type Str). The **observe** statement takes as input a probability distribution representing a likelihood (of type $D\ \sigma$), and a value representing an observation (of type σ). A generative program can include many calls to **sample** and **observe**, ultimately inducing an *unnormalized joint distribution* over *traces*: finite dictionaries mapping the names of random variables to sampled values. Intuitively, ignoring the **observe** statements in a program, we can read off a sampling distribution over traces—the *prior*. The **observe** statements then reweight each possible execution's trace by the likelihoods accumulated during that execution, yielding an unnormalized *posterior* over traces.

3.2.2 Denotational Semantics. Formally, we write $\mathbb{T}$ for the space of possible traces. It arises as a countable disjoint union, indexed by possible *trace shapes* (finite partial maps from string-valued *names* k to corresponding ground types σ_k), of product spaces $\prod_k [\![\sigma_k]\!]$. We can also define a base measure $\mathcal{B}_\mathbb{T}$ over $\mathbb{T}$, by summing product measures for each possible trace shape:

$$\mathcal{B}_\mathbb{T}(U) \coloneqq \sum_{s \subseteq \mathrm{Str} \times \Sigma} \left(\bigotimes_{(k,\sigma_k) \in s} \mathcal{B}_{\sigma_k} \right) (\{ \mathit{values}(u) \mid u \in U \wedge \mathit{shape}(u) = s \}).$$

Generative programs (of type $G\ \tau$) induce measures μ on $\mathbb{T}$ that satisfy two properties: they are absolutely continuous with respect to $\mathcal{B}_\mathbb{T}$ (and therefore have a well-defined notion of *trace density* $\frac{d\mu}{d\mathcal{B}_\mathbb{T}}$), and they are *discrete-structured*: for $(\mu \otimes \mu)$-almost-all pairs of traces (u_1, u_2), either $u_1 = u_2$, or there exists a string s present in both u_1 and u_2 such that $u_1[s] \neq u_2[s]$. In words, if two distinct traces are both in the support of a generative program, then in the two executions that those traces represent, there must have been some **sample** statement at which different choices were made for the same random variable. We write $\mathrm{Meas}^{DS}_{\ll \mathcal{B}_\mathbb{T}}\ \mathbb{T}$ for the space of measures on $\mathbb{T}$ satisfying

these two properties. The semantics then assigns $[\![G\ \tau]\!] \coloneqq \mathrm{Meas}^{DS}_{\ll \mathcal{B}_{\mathbb{T}}}\ \mathbb{T} \times (\mathbb{T} \Rightarrow [\![\tau]\!])$: a generative program denotes both a (well-behaved) measure on traces, *and* a *return-value function* that given a trace, computes the program's $[\![\tau]\!]$-valued result when its random choices are as in the trace.

The semantics for terms of type $G\ \tau$ (Fig. 4, bottom left) give a formal account of how a generative program's source code yields a particular measure on traces and return-value function. We write $[\![\cdot]\!]_i$ for $\pi_i \circ [\![\cdot]\!]$, so that $[\![\cdot]\!]_1$ computes the measure on traces and $[\![\cdot]\!]_2$ computes the return-value function. In Appx. E, we show that our term semantics really does map every program of type $G\ \tau$ a well-behaved measure over traces, i.e., the absolute continuity and discrete-structure requirements are satisfied by our definitions. The semantics of each probabilistic programming construct can be understood in terms of its trace distribution and return value function:

- **return** t: The simplest generative programs deterministically compute a return value t. These programs denote deterministic (Dirac delta) distributions on the *empty trace* $\{\}$, because they make no random choices. The return value function $[\![\textbf{return}\ t]\!]_2(\gamma)$ then maps any trace u to the program's return value, $[\![t]\!](\gamma)$.
- **sample** $t_1\ t_2$: Slightly more complicated, the **sample** $t_1\ t_2$ command denotes a measure over *singleton* traces $\{name \mapsto value\}$, namely the *pushforward* of the measure $[\![t_1]\!](\gamma)$ by the function $\lambda v.\{[\![t_2]\!](\gamma) \mapsto v\}$. The return value function for **sample** $t_1\ t_2$ statements accepts a trace u and looks up the value associated with the name $[\![t_2]\!](\gamma)$, if it exists. We define $u[name]$ to return the value associated with *name* in u, if *name* is a key in u, and otherwise to return a default value of the appropriate type.[5]
- **observe** $t_1\ t_2$: Like **return**, the statement **observe** $t_1\ t_2$ makes no random choices, and thus has an empty trace. But it denotes a *scaled* Dirac delta measure: the measure it assigns to the empty trace is equal to the density of the value $[\![t_2]\!](\gamma)$ under the measure $[\![t_1]\!](\gamma)$.
- **do**$\{x \leftarrow t; m\}$: Sequencing two generative programs concatenates their traces (which we write using the $+\!\!+$ concatenation operator). The helper $disj : \mathbb{T} \times \mathbb{T} \to \{0, 1\}$ checks whether the names used by each of two traces are distinct, returning 1 if so and 0 otherwise. We use it in our definition of $[\![\textbf{do}\{x \leftarrow t; m\}]\!]$ to model that when a program uses the same name twice in a single execution, a runtime error is raised: the semantics assigns measure 0 to those executions, leaving measure < 1 for the remaining, valid executions.[6]

3.3 Differentiable Probabilistic Programming with $\lambda_{\textbf{ADEV}}$

3.3.1 Syntax. The right panel of Fig. 4 presents a separate extension of the shared core, λ_{ADEV}, a lower-level language for differentiable probabilistic programming [40]. Like λ_{Gen}, λ_{ADEV} adds a monadic type for probabilistic computations, $P\ \tau$, which supports the sampling of primitive distributions (with **sample**), deterministic computation (with **return**), scoring by multiplicative density factors (**score**), and sequencing (with **do**). But λ_{ADEV} probabilistic programs do not denote distributions on *traces*, and the **sample** statements do not specify names for random variables. Rather, programs directly denote (quasi-Borel) measures on output types τ.

[5] All ground types σ are inhabited, and we can choose the default values arbitrarily.

[6] The semantics of a runtime error are equivalent to the semantics of **observe**-ing an impossible outcome. Because of this, if the user's model contains such errors, variational inference can be seen as training a guide program to approximate the model posterior *given* that no errors are encountered. If the variational program itself contains either **observe** statements or runtime errors, it can also denote an unnormalized measure, in which case an objective like the ELBO $(\int Q(dx) \log \frac{Zp(x)}{q(x)})$ can no longer be interpreted as a lower bound on the model's log normalizing constant $\log Z$. Our system will still produce unbiased gradient estimates for the objective, but it is likely not an objective that the user intended to optimize. This suggests that better static checks for whether a program is normalized could be useful, helping users to avoid silent optimization failures if they accidentally encode an unnormalized variational family.

Even so, we do include syntax for constructing and manipulating traces *as data*. This is because, in §4, we will develop program transformations that turn λ_{Gen} programs into λ_{ADEV} programs that (differentiably) simulate and evaluate densities of reified trace data structures.

The language also features an *expected value operator* $\mathbb{E}$, of type $P\,\mathbb{R} \to \widetilde{\mathbb{R}}$. Terms of type $\widetilde{\mathbb{R}}$ intuitively represent "losses (i.e., objectives) that can be unbiasedly estimated." The ultimate goal of a user in λ_{ADEV} is to write an objective function of type $\mathbb{R}^n \to \widetilde{\mathbb{R}}$ (where $\mathbb{R}^n = \mathbb{R} \times \cdots \times \mathbb{R}$), and then use automatic differentiation of expected values (§5) to obtain a gradient estimator for the loss function it denotes. These loss functions can be constructed by taking expectations of probabilistic programs (using $\mathbb{E}$) or by composing existing losses using new primitives (e.g., $+_{\widetilde{\mathbb{R}}}$, $\times_{\widetilde{\mathbb{R}}}$, and $\exp_{\widetilde{\mathbb{R}}}$). Example λ_{ADEV} programs include the objectives defined in §2 (ELBO, IWELBO, and IWHVI), as well as the automatically compiled programs on the right-hand side of Fig. 2.

3.3.2 Denotational Semantics. Semantically, $P\,\tau$ is the space of quasi-Borel measures on $[\![\tau]\!]$. The type $\widetilde{\mathbb{R}}$ has the same denotation as $P\,\mathbb{R}$, but cannot be used monadically (i.e., it composes in a more restricted way than $P\,\mathbb{R}$). This is because it is intended to represent an unbiased estimator of a particular real number. For example, suppose $p : P\,\mathbb{R}$ denotes a probability measure over the reals with expected value 0. Then $\mathbb{E}\,p : \widetilde{\mathbb{R}}$ denotes the same probability measure, with expected value 0. But p can be used within larger probabilistic programs; for example, we can write $p_* = \mathbf{do}\{x \leftarrow p; \mathbf{return}\,\exp(p)\}$, which draws a sample from p and exponentiates it. By Jensen's inequality, the expected value of $[\![p_*]\!]$ will generally be *greater than* $e^0 = 1$. By contrast, the term $\mathbb{E}\,p$ cannot be freely sampled within probabilistic programs, but can be composed with certain special arithmetic operators, so we can write (for example) $p^* = \exp_{\widetilde{\mathbb{R}}}(\mathbb{E}\,p)$. The primitive $\exp_{\widetilde{\mathbb{R}}}$ uses special logic to construct an unbiased estimator of e^x given an unbiased estimator of x, so the program p^* denotes a probability distribution that *does* have expectation $e^0 = 1$.

4 COMPILING DIFFERENTIABLE SIMULATORS AND DENSITY EVALUATORS

We now show how to take λ_{Gen} programs and automatically compile them into λ_{ADEV} programs that simulate traces or compute density functions. We introduce two program transformations, **sim** and **density**, which take λ_{Gen} terms to λ_{ADEV} terms that implement the desired functionality. The intuition for these transformations is as follows:

- (**sim**$\{\cdot\}$) At a high level, to simulate traces, we just run the generative program and record the value of every sample we take into a growing trace data structure.[7]
- (**density**$\{\cdot\}$) To compute the density of a trace, we execute the program, but fix the value of every primitive **sample** to equal the recorded value from the given trace, and multiply a running joint density by the density for the primitive. For **observe**, we multiply the running joint density by the density of the primitive evaluated at the value provided to the **observe** statement.

For example, simulating from the unnormalized model in Fig. 2 and then evaluating the density of the sampled trace — using **sim** and **density** as introduced in Fig. 5 — might produce:

$$\mathbf{sim}\{\texttt{model}\} \rightsquigarrow (\{\text{“}x\text{”} : 0.75, \text{“}y\text{”} : -2.2\}, 1.3 \times 10^{-4}), \quad \mathbf{density}\{\texttt{model}\}(\{\text{“}x\text{”} : 0.75, \text{“}y\text{”} : -2.2\}) = 1.3 \times 10^{-4}$$

Note that although they are not usually formalized as program transformations or rigorously proven correct, the techniques we describe here are well-known and widely used in the implementations of PPLs, e.g. in Gen, ProbTorch, and Pyro.

[7] In the full specification for **sim**, along with the simulated trace, the transformed term also returns (with probability 1) the density of the term evaluated at the simulated trace.

Spec

$$\textbf{Syntax}\ \vdash t : \sigma \to G\,\tau \implies \vdash \mathbf{density}\{t\} : \sigma \to \mathrm{Trace} \to \mathbb{R} \qquad \textbf{Semantics}\ [\![\mathbf{density}\{t\}]\!](\theta) = \frac{d[\![t]\!]_1(\theta)}{d\mathcal{B}_{\mathbb{T}}}$$

Wrapper

$$\mathbf{density}\{t\} := \lambda\theta.\lambda u.\mathbf{let}\ (x, w, u') = \xi\{t\}(\theta)(u)\ \mathbf{in}\ \mathbf{if}\ \mathit{isempty}(u')\ \mathbf{then}\ w\ \mathbf{else}\ 0$$

Helper ξ

$$\textbf{Syntax}\ \Gamma \vdash t : \tau \implies \xi\{\Gamma\} \vdash \xi\{t\} : \xi\{\tau\} \qquad \textbf{Semantics}\ \forall(\gamma,\gamma') \in R^{\xi}_{\Gamma}, ([\![t]\!](\gamma), [\![\xi\{t\}]\!](\gamma')) \in R^{\xi}_{\tau}$$

on types

$$\begin{array}{ll|ll}
\xi\{\sigma\} & := \sigma & R^{\xi}_{\sigma} & := \{(x,x) \mid x \in [\![\sigma]\!]\} \\
\xi\{D\ \sigma\} & := \sigma \to \mathbb{R} & R^{\xi}_{D\ \sigma} & := \{(\mu,\rho) \mid \rho = \frac{d\mu}{d\mathcal{B}_\sigma}\} \\
\xi\{\tau_1 \times \tau_2\} & := \xi\{\tau_1\} \times \xi\{\tau_2\} & R^{\xi}_{\tau_1\times\tau_2} & := \{((x,y),(x',y')) \mid (x,x') \in R^{\xi}_{\tau_1} \wedge (y,y') \in R^{\xi}_{\tau_2}\} \\
\xi\{\tau_1 \to \tau_2\} & := \xi\{\tau_1\} \to \xi\{\tau_2\} & R^{\xi}_{\tau_1\to\tau_2} & := \{(f,g) \mid \forall(x,y) \in R^{\xi}_{\tau_1}, (f(x),g(y)) \in R^{\xi}_{\tau_2}\} \\
\xi\{G\ \tau\} & := \mathrm{Trace} \to \xi\{\tau\} \times \mathbb{R} \times \mathrm{Trace} & R^{\xi}_{G\ \tau} & := \{(((u,f),g) \mid \exists h.\forall u \in \mathbb{T}.(f(u),h(u)) \in R^{\xi}_{\tau} \wedge \\
 & & & \quad g = \lambda u.\mathbf{let}\ (u_1,u_2) = \mathit{split}_\mu(u)\ \mathbf{in}(h(u), \tfrac{d\mu}{d\mathcal{B}_{\mathbb{T}}}(u_1), u_2)\}
\end{array}$$

on terms

$$\begin{array}{ll|ll}
\xi\{()\} & := () & \xi\{x\} & := x \\
\xi\{\lambda x.t\} & := \lambda x.\xi\{t\} & \xi\{t_1\ t_2\} & := \xi\{t_1\}\ \xi\{t_2\} \\
\xi\{(t_1,t_2)\} & := (\xi\{t_1\}, \xi\{t_2\}) & \xi\{\pi_i\ t\} & := \pi_i\ \xi\{t\} \\
\xi\{b\}(b \in \{\mathrm{T},\mathrm{F}\}) & := b & \xi\{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\} & := \mathbf{if}\ \xi\{t\}\ \mathbf{then}\ \xi\{t_1\}\ \mathbf{else}\ \xi\{t_2\} \\
\xi\{\mathbf{normal}_{strat}\} & := \lambda\mu.\lambda\sigma.\lambda x.\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} & \xi\{\mathbf{flip}_{strat}\} & := \lambda p.\lambda b.\mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ 1-p \\
\xi\{\mathbf{return}\ t\} & := \lambda u.(\xi\{t\}, 1, u) & \xi\{\mathbf{observe}\ t_1\ t_2\} & := \lambda u.((), \xi\{t_1\}(\xi\{t_2\}), u) \\
\xi\{\mathbf{sample}\ t_1\ t_2\} & := \lambda u.\mathbf{let}\ (v,w,u') = \mathit{pop}\ u\ \xi\{t_2\}\ \mathbf{in}\ (v, w \cdot \xi\{t_1\}(v), u') & \xi\{\mathbf{do}\ \{x \leftarrow t; m\}\} & := \lambda u.\mathbf{let}\ (x,w,u') = \xi\{t\}(u)\ \mathbf{in}\ \mathbf{let}\ (y,v,u'') = \xi\{\mathbf{do}\{m\}\}(u')\ \mathbf{in}\ (y, w\cdot v, u'')
\end{array}$$

Spec

$$\textbf{Syntax}\ \vdash t : \sigma \to G\,\tau \implies \vdash \mathbf{sim}\{t\} : \sigma \to P\ (\mathrm{Trace} \times \mathbb{R}) \qquad \textbf{Semantics}\ [\![\mathbf{sim}\{t\}]\!](\theta) = (\mathrm{id} \otimes \tfrac{d[\![t]\!]_1(\theta)}{d\mathcal{B}_{\mathbb{T}}})_*[\![t]\!]_1(\theta)$$

Wrapper

$$\mathbf{sim}\{t\} := \lambda\theta.\mathbf{do}\{(x,w,u) \leftarrow \chi\{t\}(\theta); \mathbf{return}\ (u,w)\}$$

Helper χ

$$\textbf{Syntax}\ \Gamma \vdash t : \tau \implies \chi\{\Gamma\} \vdash \chi\{t\} : \chi\{\tau\} \qquad \textbf{Semantics}\ \forall(\gamma,\gamma') \in R^{\chi}_{\Gamma}, ([\![t]\!](\gamma), [\![\chi\{t\}]\!](\gamma')) \in R^{\chi}_{\tau}$$

on types

$$\begin{array}{ll|ll}
\chi\{\sigma\} & := \sigma & R^{\chi}_{\sigma} & := \{(x,x) \mid x \in [\![\sigma]\!]\} \\
\chi\{D\ \sigma\} & := P\ (\sigma \times \mathbb{R}) \times (\sigma \to \mathbb{R}) & R^{\chi}_{D\ \sigma} & := \{(\mu,(\nu,\rho)) \mid \nu(U) = \int \mu(du)\delta_{\left(u, \frac{d\mu}{d\mathcal{B}_\sigma}(u)\right)}(U) \wedge \rho = \tfrac{d\mu}{d\mathcal{B}_\sigma}\} \\
\chi\{\tau_1 \times \tau_2\} & := \chi\{\tau_1\} \times \chi\{\tau_2\} & R^{\chi}_{\tau_1\times\tau_2} & := \{((x,y),(x',y')) \mid (x,x') \in R^{\chi}_{\tau_1} \wedge (y,y') \in R^{\chi}_{\tau_2}\} \\
\chi\{\tau_1 \to \tau_2\} & := \chi\{\tau_1\} \to \chi\{\tau_2\} & R^{\chi}_{\tau_1\to\tau_2} & := \{(f,g) \mid \forall(x,y) \in R^{\chi}_{\tau_1}, (f(x),g(y)) \in R^{\chi}_{\tau_2}\} \\
\chi\{G\ \tau\} & := P\ (\chi\{\tau\} \times \mathbb{R} \times \mathrm{Trace}) & R^{\chi}_{G\ \tau} & := \{(((\mu,f),\nu) \mid \exists g.\forall u.(f(u),g(u)) \in R^{\xi}_{\tau} \wedge \\
 & & & \quad \nu = \lambda U.\int \mu(du) 1_U(g(u), \tfrac{d\mu}{d\mathcal{B}_{\mathbb{T}}}(u), u)\}
\end{array}$$

on terms

$$\begin{array}{ll|ll}
\chi\{()\} & := () & \chi\{x\} & := x \\
\chi\{\lambda x.t\} & := \lambda x.\chi\{t\} & \chi\{t_1\ t_2\} & := \chi\{t_1\}\ \chi\{t_2\} \\
\chi\{(t_1,t_2)\} & := (\chi\{t_1\}, \chi\{t_2\}) & \chi\{\pi_i\ t\} & := \pi_i\ \chi\{t\} \\
\chi\{b\}(b \in \{\mathrm{T},\mathrm{F}\}) & := b & \chi\{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\} & := \mathbf{if}\ \chi\{t\}\ \mathbf{then}\ \chi\{t_1\}\ \mathbf{else}\ \chi\{t_2\} \\
\chi\{\mathbf{normal}_{strat}\} & := \lambda(\mu,\sigma).(\mathbf{do}\{ x \leftarrow \mathbf{sample}\ \mathbf{normal}_{strat}(\mu,\sigma); \mathbf{let}\ \rho = \tfrac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}; \mathbf{return}(x,\rho) \}, \lambda x.\tfrac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}) & \chi\{\mathbf{flip}_{strat}\} & := \lambda p.(\mathbf{do}\{ b \leftarrow \mathbf{sample}\ \mathbf{flip}_{strat}(p); \mathbf{let}\ \rho = \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ 1-p; \mathbf{return}(b,\rho) \}, \lambda b.\mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ 1-p) \\
\chi\{\mathbf{return}\ t\} & := \mathbf{return}(\chi\{t\}, 1, \{\}) & \chi\{\mathbf{observe}\ t_1\ t_2\} & := \mathbf{do}\{ \mathbf{let} w = \pi_2(\chi\{t_1\})(\chi\{t_2\}); \mathbf{score}\ w; \mathbf{return}\ ((), w, \{\}) \} \\
\chi\{\mathbf{sample}\ t_1\ t_2\} & := \mathbf{do}\{ (x,w) \leftarrow \pi_1(\chi\{t_1\}); \mathbf{return}(x, w, \{\chi\{t_2\} \mapsto x\}) \} & \chi\{\mathbf{do}\ \{x \leftarrow t; m\}\} & := \mathbf{do}\{ (x,w,u_1) \leftarrow \chi\{t\}; (y,v,u_2) \leftarrow \chi\{\mathbf{do}\{m\}\}; \mathbf{if}\, \mathit{disj}(u_1,u_2)\ \mathbf{then}\ \mathbf{return}\ (y, w\cdot v, u_1 +\!\!+ u_2)\ \mathbf{else}\ \mathbf{do}\{\mathbf{score}\ 0; \mathbf{return}\ (y,0,\{\})\} \}
\end{array}$$

Fig. 5. Traced simulation and density evaluation as program transformations from λ_{Gen} to λ_{ADEV}.

Differentiability Properties of Densities. In the context of variational inference, density functions must satisfy certain differentiability properties—with respect to the parameters being learned, and possibly with respect to the location at which the density is being queried, depending on the gradient estimators one wishes to apply. Previous work has developed specialized static analyses to determine smoothness properties of different parts of programs, in order to reason about gradient estimation for variational inference [34]. A benefit of our approach is that it greatly simplifies this reasoning: the overall differentiability requirements for gradient estimation are enforced by the type system of the target-language (λ_{ADEV}), as in [40]. We translate well-typed source-language programs into well-typed target-language density evaluators and trace simulators, which can be composed into well-typed variational objectives. This implies a non-trivial result: the restrictions that λ_{Gen} enforces on models and variational families are sufficient to ensure the necessary differentiability properties for unbiased estimation of variational objective gradients. Furthermore, these guarantees can be modularly extended to new variational objectives, gradient estimation strategies, and modeling language features. We discuss this further at the end of §5.

4.1 Compiling Differentiable Density Evaluators

The density program transformation **density** is given in Fig. 5 (top). As input, it processes a λ_{Gen} term t of type $\sigma \to G\ \tau$: a generative program that has some (ground type) parameter or input. The result of the transformation is a λ_{ADEV} term of type $\sigma \to \text{Trace} \to \mathbb{R}$, which, given a parameter θ and a trace u, computes the density of $[\![t]\!]_1(\theta)$ at u.

The transformation is only defined for source programs of type $\sigma \to G\ \tau$, but it is implemented using a helper transformation ξ that is defined at all source-language types. The intended behavior of ξ applied to a source-language term depends on the type of the term; we encode this type-dependent specification into a family of *relations* R^{ξ}_{τ} indexed by types τ (see Fig. 5). Each R^{ξ}_{τ} is a subset of $[\![\tau]\!] \times [\![\xi\{\tau\}]\!]$; if $(x, y) \in R^{\xi}_{\tau}$, it means that it is permissible for ξ to translate a term denoting x into a term denoting y. For example, $R^{\xi}_{D\ \sigma}$ relates measures on $[\![\sigma]\!]$ to their density functions with respect to $\mathcal{B}_{\sigma}$, to encode that ξ should transform primitive distribution terms into terms that compute primitive distribution densities. More interesting is the specification for ξ on full probabilistic programs, of type $G\ \tau$. Because the term under consideration may be only part of a larger program, ξ must compute not just a density, but also a return value (for use later in the program) and a *remainder* of its input trace, containing choices not consumed while processing the current term. The relation $R^{\xi}_{G\ \tau}$ encodes this intuition using the function $split_{\mu}$, which splits a trace u into two parts: the largest subtrace of u in the support of μ and the remaining subtrace, or, if no such subtrace exists, all of u and the empty trace (see Appx. E for a formal definition).

To prove that **density** works correctly, we first prove that ξ satisfies its intended specifications:

Lemma 4.1. *Let $\Gamma \vdash t : \tau$ be an open term of λ_{Gen}. Then $\xi\{\Gamma\} \vdash \xi\{t\} : \xi\{\tau\}$ is a well-typed open term of λ_{ADEV}, and $\forall (\gamma, \gamma') \in R^{\xi}_{\Gamma}, ([\![\tau]\!](\gamma), [\![\xi\{\tau\}]\!](\gamma')) \in R^{\xi}_{\tau}$.*

The proof (in Appx. E) is by induction, but because the inductive hypothesis is different at each type τ (depending on our definition of R^{ξ}_{τ}), it is an example of what is often called a *logical relations* proof. Once it is proven, we are ready to prove the main correctness theorem for densities:

Theorem 4.2. *Let $\vdash t : \sigma \to G\ \tau$ be a closed λ_{Gen} term for some ground type σ. Then $\vdash$ **density**$\{t\} : \sigma \to \text{Trace} \to \mathbb{R}$ is a well-typed λ_{ADEV} term and for all $\theta \in [\![\sigma]\!]$, $[\![$**density**$\{t\}]\!](\theta)$ is a density function for $\pi_1([\![t]\!])(\theta)$ with respect to $\mathcal{B}_{\mathbb{T}}$.*

Proof. Fix $\theta \in [\![\sigma]\!]$ and let $(\mu, f) = [\![t]\!](\theta)$. By Lemma 4.1, we have that $([\![t]\!](\theta), [\![\xi\{t\}]\!](\theta)) \in R^{\xi}_{G\ \tau}$. Now consider a trace $u \in \mathbb{T}$. The **density** macro invokes $\xi\{t\}\ \theta$ on u to obtain a triple (x, w, u').

By the definition of $R^{\xi}_{G\ \tau}$, we have that $w = \frac{d\mu}{d\mathcal{B}_{\mathbb{T}}}(u_1)$ and $u' = u_2$, for $(u_1, u_2) = split_{\mu}(u)$. Recall that if u is in the support of μ, *or* if *no subtrace* of u is in the support of μ, then $split_{\mu}$ returns $(u, \{\})$, causing **density** to enter its **then** branch and return $w = \frac{d\mu}{d\mathcal{B}_{\mathbb{T}}}(u_1) = \frac{d\mu}{d\mathcal{B}_{\mathbb{T}}}(u)$ as desired. If u is not in the support of μ but has a subtrace that is, then $split_{\mu}$ returns that subtrace, along with a non-empty u_2. In this case the **else** branch of **density** correctly returns 0 (because u is not in the support). □

4.2 Compiling Differentiable Trace Simulators

The simulation program transformation **sim** is given in Fig. 5 (bottom). Like **density**, it processes as input a λ_{Gen} term t of type $\sigma \to G\ \tau$, but it generates a term of type $P\ (\text{Trace} \times \mathbb{R})$, satisfying the specification that the pushforward by π_1 is the original program's measure over traces, $[\![t]\!]_1(\theta)$, and that with probability 1, the second component is the density of $[\![t]\!]_1$ evaluated at the sampled trace. Like **density**, **sim** is implemented using a helper macro χ defined at all types. Fig. 5 presents the logical relations R^{χ}_{τ} specifying the helper's intended behavior on terms of type τ. These relations are simpler than those for ξ; for example, on terms of type $G\ \tau$, χ has almost the same specification as **sim** itself, except that it must also compute a return value.

One feature of χ that is worth noting is its translations of primitives. If a primitive used within a traced probabilistic program is annotated with a gradient estimation strategy, then the translated program uses the same annotated primitive, and then computes a density. This is only well-typed because the *density functions* of primitives that return $\mathbb{R}$ values (i.e., not $\mathbb{R}^*$ values) are smooth. This is not a requirement of ADEV in general, but we require it in order to automate differentiable traced simulation.

To prove correctness, we again begin by showing the helper is sound:

Lemma 4.3. *Let $\Gamma \vdash t : \tau$ be an open term of λ_{Gen}. Then $\chi\{\Gamma\} \vdash \chi\{t\} : \chi\{\tau\}$ is a well-typed open term of λ_{ADEV}, and $\forall(\gamma, \gamma') \in R^{\chi}_{\Gamma}, ([\![\tau]\!](\gamma), [\![\chi\{\tau\}]\!](\gamma') \in R^{\chi}_{\tau}$.*

The proof is again by logical relations, and can be found in Appx. E. We can then prove the correctness of **sim** itself:

Theorem 4.4. *Let $\vdash t : G\ \tau$ be a closed λ_{Gen} term. Then $\vdash \mathbf{sim}\{t\} : P\ (Trace \times \mathbb{R})$ is a well-typed λ_{ADEV} term and $[\![\mathbf{sim}\{t\}]\!]$ is the pushforward of $\pi_1([\![t]\!])$ by the function $\lambda u. \left(u, \frac{d[\![t]\!]_1}{d\mathcal{B}_{\mathbb{T}}}(u)\right)$.*

Proof. Fix $\theta \in [\![\sigma]\!]$ and let $(\mu, f) = [\![t]\!](\theta)$. By Lemma 4.3, we have that $([\![t]\!](\theta), [\![\chi\{t\}]\!](\theta)) \in R^{\chi}_{G\ \tau}$. The **sim** macro invokes $\chi\{t\}\theta$ to obtain a triple (x, w, u), but only returns (u, w). Observe that the requirements placed by $R^{\chi}_{G\ \tau}$ on w and u are precisely the conditions we aim to prove here. □

5 VARIATIONAL INFERENCE VIA DIFFERENTIABLE PROBABILISTIC PROGRAMMING

As we saw in §2, the density and trace simulation programs automated in the previous section can be used to construct larger λ_{ADEV} programs implementing variational objectives. Once we have a λ_{ADEV} program representing our objective function, we need to differentiate it. Conventional AD systems do not correctly handle randomness in objective functions, or the expectation operator $\mathbb{E}$, and will produce biased gradient estimators when applied naively [40]. For example, standard AD has no way of propagating derivative information through a primitive like flip : $[0, 1] \to P\ \mathbb{B}$ (to do so, one would need to define the notion of *derivative of a Boolean with respect to the probability that it was heads*). The ADEV algorithm [40] is designed to handle these features, and can be used to derive unbiased gradient estimators automatically.

Extending ADEV with Traces and Unnormalized Measures. Fig. 6 gives the ADEV program transformation, extended to handle new datatypes (traces) and unnormalized measures (due to

Syntax $\vdash t : \mathbb{R}^n \to \widetilde{\mathbb{R}} \implies \vdash \mathbf{adev}\{t\} : \mathbb{R}^n \to \mathbb{R}^n \to \widetilde{\mathbb{R}}$
$\vdash t : \mathbb{R}^n \to \widetilde{\mathbb{R}} \implies \vdash \mathbf{dom}\{t\}_{(\theta,i)} : \mathbb{R} \times \mathbb{R}^* \to \mathbb{R}$

Semantics $(\forall \theta \in \mathbb{R}^n, i \in \underline{n}.[\![\mathbf{dom}\{t\}_{(\theta,i)}]\!]$ locally dom'd$) \implies \mathbb{E}_{x \sim [\![\mathbf{adev}\{t\}]\!](\theta,\mathbf{v})}[x] = \left(\nabla_\theta \int_{\mathbb{R}} x\, [\![t]\!](\theta, dx)\right)^T \mathbf{v}$

$\mathbf{adev}\{t\} \coloneqq \lambda\theta.\lambda\mathbf{v}.\mathbb{E}(\mathbf{do}\{(y, y') \leftarrow \mathcal{D}\{t\}\ ((\theta_1, \mathbf{v}_1), \ldots, (\theta_n, \mathbf{v}_\mathbf{n})); \mathbf{return}\ y'\})$
$\mathbf{dom}\{t\}_{(\theta,i)} \coloneqq \lambda(\phi, x).\pi_2(\pi_2(\mathcal{D}\{t\}((\theta_1, 0), \ldots, (\theta_{i-1}, 0), (\phi, 1), (\theta_{i+1}, 0), \ldots, (\theta_n, 0))\ x$

Syntax $\Gamma \vdash_{\text{ADEV}} t : \tau \implies \mathcal{D}\{\Gamma\} \vdash_{\text{ADEV}} \mathcal{D}\{t\} : \mathcal{D}\{\tau\}$ ***Semantics*** $\forall (\gamma, \gamma') \in R^{\mathcal{D}}_{\Gamma}, ([\![t]\!] \circ \gamma, [\![\mathcal{D}\{t\}]\!] \circ \gamma') \in R^{\mathcal{D}}_{\tau}$

Type Translation and Logical Relations

$\mathcal{D}\{\mathbb{R}\} \coloneqq \mathbb{R} \times \mathbb{R}$
$\mathcal{D}\{\sigma\} \coloneqq \sigma \quad (\sigma \in \{\mathbb{R}^*, \mathbb{B}, \text{Str}\})$
$\mathcal{D}\{\tau_1 \times \tau_2\} \coloneqq \mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\}$
$\mathcal{D}\{\tau_1 \to \tau_2\} \coloneqq \mathcal{D}\{\tau_1\} \to \mathcal{D}\{\tau_2\}$
$\mathcal{D}\{\text{Trace}\} \coloneqq \text{Trace}$
$\mathcal{D}\{D\ \sigma\} \coloneqq \mathcal{D}\{P\ \sigma\}$
$\mathcal{D}\{\widetilde{\mathbb{R}}\} \coloneqq P\ (\mathbb{R} \times \mathbb{R}) \times (\mathbb{R}^* \to \mathbb{R} \times \mathbb{R})$
$\mathcal{D}\{P\ \tau\} \coloneqq (\mathcal{D}\{\tau\} \to \mathcal{D}\{\widetilde{\mathbb{R}}\}) \to \mathcal{D}\{\widetilde{\mathbb{R}}\}$

$R^{\mathcal{D}}_{\mathbb{R}} \coloneqq \{(f, g) \mid g = \lambda r.(f(r), f'(r))\}$
$R^{\mathcal{D}}_{\sigma} \coloneqq \{(f, f) \mid f \text{ constant}\}$
$R^{\mathcal{D}}_{\tau_1 \times \tau_2} \coloneqq \{(f, g) \mid \forall i \in \{1, 2\}.(\pi_i \circ f, \pi_i \circ g) \in R^{\mathcal{D}}_{\tau_i}\}$
$R^{\mathcal{D}}_{\tau_1 \to \tau_2} \coloneqq \{(f, g) \mid \forall (x, y) \in R^{\mathcal{D}}_{\tau_1}, (\lambda r.f(r)(x(r)), g(r)(y(r))) \in R^{\mathcal{D}}_{\tau_2}\}$
$R^{\mathcal{D}}_{\text{Trace}} \coloneqq \{(f, g) \mid \forall k \in [\![\text{Str}]\!].(\lambda r.f(r)[k], \lambda r.g(r)[k]) \in R^{\mathcal{D}}_{\sigma}\}$
$R^{\mathcal{D}}_{D\ \sigma} \coloneqq R^{\mathcal{D}}_{P\ \sigma}$
$R^{\mathcal{D}}_{\widetilde{\mathbb{R}}} \coloneqq \{(\mu, \nu) \mid \forall \theta.\ \int_{\mathbb{R}} h_1(\theta)(s)ds = \int_{\mathbb{R}} x\mu(\theta, dx) = \mathbb{E}_{x \sim \pi_{1*}(\pi_1 \circ \nu)(\theta)}[x] \wedge \forall \theta.\ \int_{\mathbb{R}} h_2(\theta)(s)ds = \mathbb{E}_{x \sim \pi_{2*}(\pi_{1*}\nu)(\theta)}[x] \wedge (\lambda\theta.\lambda s.h_1(\theta)(s), \lambda\theta.\lambda s.(h_1(\theta)(s), h_2(\theta)(s)) \in R_{\mathbb{R}^* \to \mathbb{R}}$
where $h_i \coloneqq \lambda\theta.\lambda s.\pi_i((\pi_2 \circ \nu)(\theta)(s))$
$R^{\mathcal{D}}_{P\ \tau} \coloneqq \{(\mu, \nu) \mid (\lambda r.\lambda k.\lambda U.\mathbb{E}_{x \sim \mu(r)}[k(x, U)], \nu) \in R^{\mathcal{D}}_{(\tau \to \widetilde{\mathbb{R}}) \to \widetilde{\mathbb{R}}}\}$

Term Translation

$\mathcal{D}\{()\} \coloneqq ()$
$\mathcal{D}\{c\} \coloneqq c_{\mathcal{D}}$
$\mathcal{D}\{\lambda x.t\} \coloneqq \lambda x.\mathcal{D}\{t\}$
$\mathcal{D}\{(t_1, t_2)\} \coloneqq (\mathcal{D}\{t_1\}, \mathcal{D}\{t_2\})$
$\mathcal{D}\{b\}\ (b \in \{\mathbf{T}, \mathbf{F}\}) \coloneqq b$
$\mathcal{D}\{\{t_1 \mapsto t_2\}\} \coloneqq \{\mathcal{D}\{t_1\} \mapsto \mathcal{D}\{t_2\}\}$
$\mathcal{D}\{t_1[t_2]\} \coloneqq \mathcal{D}\{t_1\}[\mathcal{D}\{t_2\}]$
$\mathcal{D}\{r\} \coloneqq (r, 0)$

$\mathcal{D}\{\{\}\} \coloneqq \{\}$
$\mathcal{D}\{x\} \coloneqq x$
$\mathcal{D}\{t_1\ t_2\} \coloneqq \mathcal{D}\{t_1\}\ \mathcal{D}\{t_2\}$
$\mathcal{D}\{\pi_i\ t\} \coloneqq \pi_i\ \mathcal{D}\{t\}$
$\mathcal{D}\{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\} \coloneqq \mathbf{if}\ \mathcal{D}\{t\}\ \mathbf{then}\ \mathcal{D}\{t_1\}\ \mathbf{else}\ \mathcal{D}\{t_2\}$
$\mathcal{D}\{t_1 + t_2\} \coloneqq \mathcal{D}\{t_1\} + \mathcal{D}\{t_2\}$
$\mathcal{D}\{\mathbf{sample}\ t\} \coloneqq \mathcal{D}\{t\}$
$\mathcal{D}\{\mathbf{score}\ t\} \coloneqq \lambda\kappa.(\mathbf{do}\{y \leftarrow \pi_1(\kappa(()));\ \mathbf{return}(\mathcal{D}\{t\} \times_{\mathcal{D}} y)\}, \lambda s.(\pi_2(\kappa(()))\ s) \times_{\mathcal{D}} \mathcal{D}\{t\})$

$\mathcal{D}\{\mathbf{return}\ t\} \coloneqq \lambda\kappa.\kappa(\mathcal{D}\{t\})$
$\mathcal{D}\{\mathbf{do}\ \{x \leftarrow t; m\}\} \coloneqq \lambda\kappa.\mathcal{D}\{t\}(\lambda x.\mathcal{D}\{\mathbf{do}\{m\}\}(\kappa))$
$\mathcal{D}\{\mathbb{E}\ t\} \coloneqq \mathcal{D}\{t\}\ (\lambda x.(\mathbf{return}\ x, \lambda s.x))$

$\mathcal{D}\{\mathbf{normal}_{\text{REPARAM}}\} \coloneqq \lambda((\mu, \mu'), (\sigma, \sigma')).\lambda\kappa.(\mathbf{do}\{\epsilon \leftarrow \mathbf{sample}\ (\mathbf{normal}_{\text{REPARAM}}(0, 1));\ \pi_1(\kappa((\sigma\epsilon + \mu, \sigma'\epsilon + \mu')))\}, \lambda s.\ldots.)$

$\mathcal{D}\{\mathbf{normal}_{\text{REINFORCE}}\} \coloneqq \lambda((\mu, \mu'), (\sigma, \sigma')).\lambda\kappa.(\mathbf{do}\{x \leftarrow \mathbf{sample}\ (\mathbf{normal}_{\text{REINFORCE}}(\mu, \sigma));\ (y, y') \leftarrow \pi_1(\kappa((x, 0)));\ \mathbf{let}\ l' = \sigma'(\frac{1}{\sigma} + \frac{(y-\mu)^2}{\sigma^3}) + \mu'\frac{y-\mu}{\sigma^2};\ \mathbf{return}\ (y, y' + yl')\}, \lambda s.\ldots.)$

$\mathcal{D}\{\mathbf{flip}_{\text{ENUM}}\} \coloneqq \lambda(p, p').\lambda\kappa.(\mathbf{do}\{(y_T, y'_T) \leftarrow \pi_1(\kappa(\mathbf{T}));\ (y_F, y'_F) \leftarrow \pi_1(\kappa(\mathbf{F}));\ \mathbf{let}\ y = py_T + (1-p)y_F;\ \mathbf{let}\ y'_1 = p'y_T + py'_T;\ \mathbf{let}\ y'_2 = (1-p)y'_F - p'y_F;\ \mathbf{return}\ (y, y'_1 + y'_2)\}, \lambda s.\ldots.)$

$\mathcal{D}\{\mathbf{flip}_{\text{REINFORCE}}\} \coloneqq \lambda(p, p').\lambda\kappa.(\mathbf{do}\{b \leftarrow \mathbf{sample}\ (\mathbf{flip}_{\text{REINFORCE}}(p));\ (y, y') \leftarrow \pi_1(\kappa(b));\ \mathbf{let}\ l' = \mathbf{if}\ b\ \mathbf{then}\ \frac{p'}{p}\ \mathbf{else}\ \frac{p'}{p-1};\ \mathbf{return}(y, y' + yl')\}, \lambda s.\ldots.)$

Fig. 6. Monte Carlo gradient estimation as a program transformation from λ_{ADEV} to λ_{ADEV}.

score). Fig. 6 shows two top-level transformations, **adev** and **dom**. The **dom** transformation exists solely for analytical purposes, to produce a term that must satisfy a *local domination* condition in order for gradient estimates to be unbiased:[8]

Definition 5.1 (locally dominated). A function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is locally dominated if, for every $\theta \in \mathbb{R}$, there is a neighborhood $U(\theta) \subseteq \mathbb{R}$ of θ and an integrable function $m_{U(\theta)} : \mathbb{R} \to [0, \infty)$ such that $\forall \theta' \in U(\theta), \forall x \in \mathbb{R}, |f(\theta', x)| \leq m_{U(\theta)}(x)$.

[8]We have omitted several terms from Fig. 6, denoted with "...". These terms are as in [40, Fig. 26], and do not affect the behavior of the **adev** transformation, only **dom**.

Under this mild assumption, ADEV produces correct unbiased gradient estimators:

THEOREM 5.2. *Let* $\vdash t : \mathbb{R}^n \to \widetilde{\mathbb{R}}$ *be a closed* λ_{ADEV} *term, satisfying the following preconditions:*
(1) $\int_{\mathbb{R}} x \, [\![t]\!](\theta, dx)$ *is finite for every* $\theta \in \mathbb{R}^n$.
(2) $[\![\mathbf{dom}\{t\}_{(\theta,i)}]\!]$ *is locally dominated for every* $\theta \in \mathbb{R}^n$ *and* $i \in \underline{n}$.

Then $\vdash \mathbf{adev}\{t\} : \mathbb{R}^n \to \mathbb{R}^n \to \widetilde{\mathbb{R}}$ *is a well-typed* λ_{ADEV} *term, satisfying the following properties:*

- $[\![\mathbf{adev}\{t\}]\!](\theta, \mathbf{v})$ *is a probability measure with finite expectation for all* $\theta, \mathbf{v} \in \mathbb{R}^n$.
- $[\![t]\!]$ *is differentiable and* $\mathbb{E}_{x \sim [\![\mathbf{adev}\{t\}]\!](\theta,\mathbf{v})}[x] = \left(\nabla_\theta \int_{\mathbb{R}} x \, [\![t]\!](\theta, dx)\right)^T \mathbf{v}$.

The proof, as in the previous section, is a logical relations proof, and Fig. 6 gives the logical relations. It extends the proof of [40] with new cases, for **score**, as well as for trace operations.

Static Checks and Unbiasedness. Probabilistic programming languages like Pyro and Gen use specialized logic — also going beyond ordinary AD — to unbiasedly estimate derivatives of particular variational objectives like the ELBO, for user-defined models and variational families. But in these systems, biased gradient estimates can still arise if the user's model or variational family violates assumptions made by the PPL backend. For example:

- The user's program may sample x from a normal distribution, then branch on whether $x < k$ for some constant threshold k, in order to decide on the distribution of another random variable y. Pyro's default gradient estimation strategy assumes that the joint density of the model is differentiable with respect to the values of Gaussian random variables like x, but this assumption is violated by the user's program, because the joint density $p(x, y)$ is of the form $p(x) \cdot ([x > k] \cdot p_1(y \mid x) + [x \leq k] \cdot p_2(y \mid x))$, which may be discontinuous at $x = k$.
- Gen's default gradient estimation strategy does not place differentiability assumptions on the user's program, but does assume that the support of each primitive distribution does not depend on learned parameters. If the user's program samples from a uniform distribution with learned endpoints a and b, this assumption will be violated, and Gen's gradient estimates will be biased.

In our design, by contrast, there is no default gradient estimation strategy. Rather, the user chooses a different gradient estimation strategy for each primitive, and the overall gradient estimator is automated compositionally. Crucially, different versions of primitives (employing different gradient estimation strategies) have static types that enforce the key assumptions necessary for their unbiasedness. For example:

- The $\texttt{normal}_{\text{REPARAM}}$ primitive has type $\mathbb{R} \times \mathbb{R}_{>0} \to D\,\mathbb{R}$, so if x is drawn from $\texttt{normal}_{\text{REPARAM}}$, then the type of the variable x is $\mathbb{R}$. The type of $<$ is $\mathbb{R}^* \times \mathbb{R}^* \to \mathbb{B}$, and so the expression $x < k$ is ill-typed. Thus, the types enforce that the smoothness assumptions of the reparameterization estimator hold for the user's program, if the user chooses to apply this estimator.
- In our system, the uniform distribution with custom endpoints is **uniform** $: \mathbb{R}^* \times \mathbb{R}^* \to D\,\mathbb{R}^*$, which behaves like a safe version of Gen's uniform distribution—its output can be used non-smoothly, but its bounds must not depend directly on learned parameters. (The bounds may still depend on, e.g., Gaussian random choices with learned means.)

Our smoothness-typing discipline in λ_{Gen} is similar to, but slightly different from, that in Lew et al. [40]. As an example, their version of ADEV can support a primitive $\text{uniform}_{\text{MVD}} : \mathbb{R} \times \mathbb{R} \to P\,\mathbb{R}$, but we cannot introduce such a primitive that returns $D\,\mathbb{R}^*$ or $D\,\mathbb{R}$. This is because the program transformation ξ would need to translate such primitives into code for computing the uniform distribution's *density* as a smooth function of its endpoints—which is not possible, since the density of the uniform distribution is discontinuous at the endpoints.

These static checks are necessary for proving unbiasedness, as without them, we could easily produce estimators that do not respect the restrictions of the estimation strategies they employ.

6 CORRECTNESS OF GRADIENT ESTIMATION FOR VARIATIONAL INFERENCE

We can put together the results of the previous two sections to prove a general correctness theorem for our approach to variational inference. Suppose the user has written the following three programs:

- A **model program**: a closed λ_{Gen} program $P : \mathbb{R}^n \to G\ \tau_1$.
- A **variational program**: a closed λ_{Gen} program $Q : \mathbb{R}^m \to G\ \tau_2$.
- An **objective program**: a closed λ_{ADEV} program $L : \mathbb{R}^{n+m} \to \widetilde{\mathbb{R}}$, of the form

$$L = \lambda(\theta, \phi).\mathbf{let}\ (p, \mathbb{P}, q, \mathbb{Q}) = (\mathbf{density}\{P\}\ \theta, \mathbf{sim}\{P\}\ \theta, \mathbf{density}\{Q\}\ \phi, \mathbf{sim}\{Q\}\ \phi)\ \mathbf{in}\ F,$$

where θ and ϕ do not occur free in F. This program encodes the variational objective $\mathcal{F}(\mu_P, \mu_Q) = \int x \cdot [\![F]\!](\gamma(\mu_P, \mu_Q), dx)$, where $\gamma(\mu_P, \mu_Q)$ is an environment mapping p and q to densities of μ_P and μ_Q (with respect to $\mathcal{B}_{\mathbb{T}}$) and $\mathbb{P}$ and $\mathbb{Q}$ to simulators for μ_P and μ_Q.

The user wants to find θ and ϕ that optimize $\mathcal{F}([\![P]\!]_1(\theta), [\![Q]\!]_1(\phi))$. Our result is that our system estimates derivatives of this objective unbiasedly, under mild technical conditions:

Theorem 6.1. *Let $\mathcal{L}(\theta, \phi) = \mathcal{F}([\![P]\!]_1(\theta), [\![Q]\!]_1(\phi))$. If for all $\theta \in \mathbb{R}^n$ and $\phi \in \mathbb{R}^m$, $\mathcal{L}(\theta, \phi)$ is finite and $[\![\mathbf{dom}\{L\}_{((\theta,\phi),i)}]\!]$ is locally dominated for each $i \in \underline{n+m}$, then for all $\mathbf{v} \in \mathbb{R}^{n+m}$, $[\![\mathbf{adev}\{L\}]\!]((\theta, \phi), \mathbf{v})$ is a probability measure with finite expectation and*

$$\mathbb{E}_{x \sim [\![\mathbf{adev}\{L\}]\!]((\theta,\phi),\mathbf{v})}[x] = (\nabla_{(\theta,\phi)}\mathcal{L}(\theta, \phi))^T\mathbf{v}.$$

To understand the guarantee the theorem gives more concretely, consider the following objective program defining the ELBO (Eqn. 3):

$$\mathbf{ELBO} \coloneqq \lambda(\theta, \phi).\mathbb{E}(\mathbf{do}\ \{(z, w_q) \leftarrow (\mathbf{sim}\{Q\}\ \phi); \mathbf{let}\ w_p = (\mathbf{density}\{P\}\ \theta)\ z; \mathbf{return} \log w_p - \log w_q\})$$

We can write it in the form required by the theorem as follows:

$$L = \lambda(\theta, \phi).\mathbf{let}\ (p, \mathbb{P}, q, \mathbb{Q}) = (\mathbf{density}\{P\}\ \theta, \mathbf{sim}\{P\}\ \theta, \mathbf{density}\{Q\}\ \phi, \mathbf{sim}\{Q\}\ \phi)\ \mathbf{in}$$
$$\mathbb{E}(\mathbf{do}\{(z, w_q) \leftarrow \mathbb{Q}; \mathbf{return}\ (\log p(z) - \log w_q)\})$$

The theorem then establishes that, under mild technical conditions, applying **adev** to L yields an unbiased estimator of $\nabla_{(\theta,\phi)}\mathcal{F}([\![P]\!]_1(\theta), [\![Q]\!]_1(\phi))$.

7 FULL SYSTEM

In the previous sections, we presented a formal model of the key features of our approach. Our full language and system (detailed in Appx. A) extends our formal model in three key ways:

- **New language features for probabilistic models and variational families (Appx. A.1).** Our full language includes constructs for *marginalizing* (**marginal**) and *normalizing* (**normalize**) λ_{Gen} programs, making it possible to express a broader class of models and variational families than in current systems. Our versions of these constructs are designed following Lew et al. [39].
- **Differentiable stochastic estimators of densities and density reciprocals (Appx. A.2).** When exact densities of λ_{Gen} programs cannot be efficiently computed, our full system can compile λ_{ADEV} terms implementing differentiable unbiased estimators of the required density functions and their reciprocals. These estimators can even have learnable parameters controlling their variance, which can be optimized jointly as part of the overall variational objective.
- **Reverse-mode automatic differentiation of expected values (Appx. A.4).** Our full language's AD algorithm computes *vector-Jacobian products* for expected values of probabilistic objectives, whereas our formal development shows only *Jacobian-vector products*. Algorithms for vector-Jacobian products, also known as *reverse-mode* AD algorithms, are much more efficient when optimizing scalar losses with large numbers of parameters, common in deep learning.

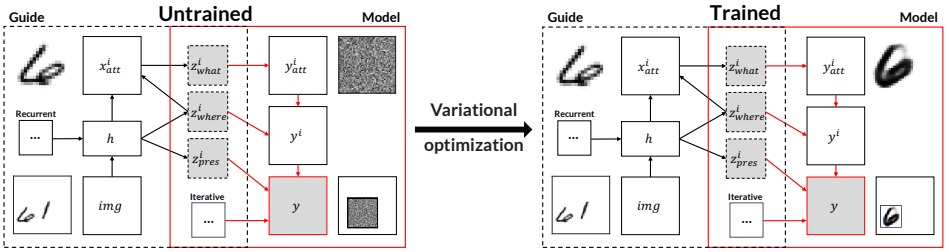

Fig. 7. AIR is a generative model for multi-object images, trained with variational inference. The model randomly selects a number of patches to render onto a canvas, and a location, scale, and latent code for each. The variational family predicts these latent variables from an image. The model is trained on a dataset of images constructed by randomly translating and scaling MNIST digits onto a canvas.

8 EVALUATION

We evaluate our approach using `genjax.vi`, a prototype of our proposed architecture implemented as an extension to a JAX-hosted version of Gen [14]. All experiments were run on a single device with an AMD Ryzen 7 7800X3D @ 5.050 GHz CPU and an Nvidia RTX 4090 GPU. We consider several case studies designed to answer the following questions:

- **Overhead.** *How much overhead is incurred by using our automated gradient estimators, over hand-coded versions?* We compare the same gradient estimator for a variational autoencoder [29] constructed (a) via a hand-coded implementation and (b) via our automation.
- **Overall performance.** *How well can we solve a challenging inference problem using our system compared to other PPLs that support variational inference?* We consider the *Attend-Infer-Repeat* (AIR) model [17] and compare the capabilities of our system to Pyro [6].
- **Expressivity and compositional correctness.** *For the objectives and estimator strategies expressible in our system, is it possible to combine all objectives and estimator strategies while maintaining correctness?* We evaluate the expressiveness of our system vs. Pyro on the AIR model, and on a hierarchical variational inference problem [1].

Overhead. Table 1 presents a runtime comparison between `genjax.vi` and a hand-coded implementation of the gradient estimator in JAX (Appx. C). We measure the wall time required to compute a gradient estimate for different batch sizes n. We find that our automation introduces a small amount of runtime overhead (around 3-10%) compared to our hand coded implementation (Fig. 10).

Overall Performance. We consider the *Attend, Infer, Repeat* [17] (AIR) model (Fig. 7). We plot accuracy and loss curves over time in Fig. 8, for several estimators expressed in our system and in Pyro. We also compare timing results, shown in Table 2. Our implementation's performance is competitive, and we support a broader class of estimators and objectives than Pyro. We find that some estimators we support (in particular those based on measure-valued derivatives) lead to faster convergence than the estimators automated by Pyro.

Table 1. Timing (ms) our estimators versus hand coded estimators for the VAE.

Batch size	Ours	Hand coded
64	0.11 ± 0.02	0.09 ± 0.04
128	0.22 ± 0.2	0.16 ± 0.08
256	0.31 ± 0.18	0.29 ± 0.17
512	0.56 ± 0.35	0.54 ± 0.34
1024	1.58 ± 1.13	1.07 ± 0.70

Table 2. Time (in seconds) to train the AIR model [17] for one epoch (batch size 64) with different objectives and estimators. All discrete variables use the same estimation strategy. IWELBO runs have $n = 2$ particles.

System	Compiler	REINFORCE	ENUM	MVD	IWELBO + REINFORCE	IWELBO + MVD
`genjax.vi`	JAX (XLA)	1.52 ± 0.05	6.22 ± 0.29	1.74 ± 0.04	2.28 ± 0.12	3.74 ± 0.05
`pyro`	Torch	12.28 ± 0.55	122.93 ± 1.74	X	22.17 ± 1.2	X

Expressivity and Compositional Correctness. In Table 3, we enumerate several possible combinations of gradient estimation strategies and objectives for the AIR model. In Table 4, we consider the model shown in Fig. 2, and implement the model and naive variational guide in `genjax.vi`, Pyro and NumPyro, as well as the auxiliary variable variational guide in `genjax.vi`. We show statistics on final mean objective values for different variational objectives. While ELBO and IWELBO are standard, our system allows using more expressive approximations and tighter lower bound objectives *compositionally* (such as DIWHVI [65], which uses SIR to estimate densities of marginals, in the IWELBO objective) to achieve tighter variational bounds.

Table 3. Combinatorial space of gradient estimators and objective functions for the AIR model, which our programmable approach helps to explore.

RE = REINFORCE estimator, EN = enumeration estimator, BL = REINFORCE with learned baselines, MV = measure-valued derivative estimator

Grad. Estimation Strategies				Objective	Batch	System	
RE.	EN.	BL.	MV.			Pyro	Ours
✓				ELBO	≥ 1	✓	✓
				IWAE	≥ 1	✓	✓
	✓			ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
		✓		ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
			✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓			ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
✓		✓		ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
✓			✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓	✓		ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓		✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
		✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓	✓		ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓		✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓		✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓	✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
Not Exploited				RWS	1	✓	✓
					> 1	×	✓

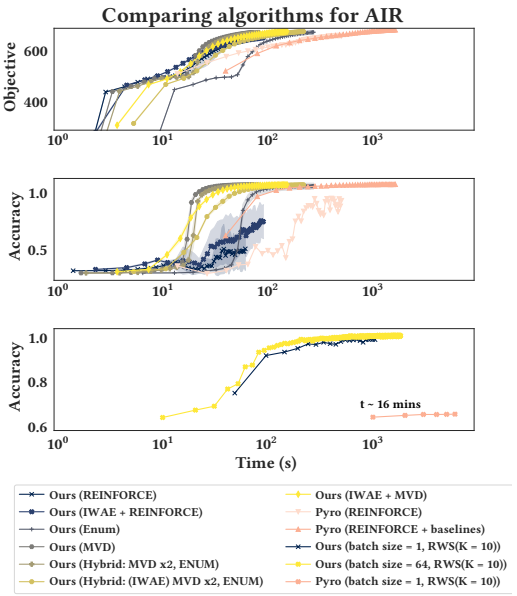

Fig. 8. We evaluate a variety of custom estimators and objectives (ELBO, IWAE, RWS) using our system. Our on average best estimator (IWAE + MVD, not expressible in Pyro) converges an order of magnitude faster than Pyro's recommended estimator.

Table 4. Mean objective value (in nats) on repeated runs for several variational objectives, including ones which utilize **marginal**. n and m denote particle sizes for SIR algorithms.

System	ELBO	IWELBO ($n = 5$)	HVI	IWHVI ($m = 5$)	DIWHVI ($n = 5, m = 5$)
`genjax.vi`	-8.08	-7.79	-9.75	-8.18	-7.33
`numpyro`	-8.08	-7.77	✓/ X	X	X
`pyro`	-8.08	-7.75	✓/ X	X	X

9 RELATED WORK

Variational Inference in PPLs. Many PPLs support some form of variational inference [6, 12, 14, 21, 67], and it is the primary focus of Pyro [6] and ProbTorch [67]. Both Pyro and ProbTorch have endeavored to make inference more programmable. For example, ProbTorch has introduced *inference combinators* that make it easy to express certain nested variational inference algorithms [67, 74]. Pyro has perhaps the most mature support for variational inference, with many gradient estimators and objectives supported. Pyro also implements some variance reduction strategies not

yet supported by our system, e.g. exploiting conditional independence using their **plate** operator. However, extending Pyro with new variational objectives or gradient estimation strategies requires a deep understanding of its internals. Furthermore, Pyro's modeling language does not have our constructs for marginalization and normalization (although Pyro can marginalize discrete, finite-support auxiliary variables from models). Concurrently with this work, Wagner [71] presented a PPL with correct-by-construction variational inference, where their notion of "correctness" is stronger in some ways and weaker in others than that of this work. In particular, Wagner [71] works with *smoothed approximations* of the user's probabilistic programs, and so the gradient estimates it computes are biased for the original objective. However, the degree of error in this smoothing is gradually annealed over the course of optimization, leading to convergence to a stationary point of the original objective.

Static Analyses for Differentiability Criteria. Within the PL community, researchers have made some progress toward formalizing and developing static analyses for ensuring soundness properties of variational inference [34, 35, 72], as well as program transformations for automatically constructing variational families informed by the model's structure [41]. By contrast, we formalize the process by which user model, inference, and objective code is transformed into a gradient estimator, tracking the interactions between density computation, simulation, and automatic differentiation. One interesting direction would be to extend Lee et al. [34]'s analysis to automatically annotate λ_{Gen} programs with gradient estimators. Note that our current type-based analysis does not verify the local domination condition of Thm. 5.2, which Wagner [71] does manage to check statically, by imposing restrictions on the PPL and considering only the ELBO objective.

Automated Gradient Estimation. Due to its centrality to many applications in computer science and beyond, there has been intense interest in automating unbiased gradient estimation for objective functions expressed as expectations, yielding several frameworks for unbiasedly differentiating first-order stochastic computation graphs [31, 63, 73], imperative programs with discrete randomness [3], and higher-order probabilistic programs [40]. Some works have also investigated automated computation of *biased* gradient estimates via smoothing [30]. However, these frameworks cannot be directly applied to variational inference problems, which require differentiating not just user code, but also traced simulators and log density evaluators of the probabilistic programs that users write. We address these challenges, by compiling density functions of user-defined probabilistic programs into a target language compatible with ADEV [40]. We also extend ADEV in several other ways: our version adds **score** to differentiate not just expected values but general integrals against (potentially unnormalized) measures; is implemented performantly on GPU; and has been extended with a reverse-mode.

Programmable Inference. We build on a long line of work that aims to make inference more *programmable* in PPLs [14, 39, 47, 48, 54]. In much the same way that this prior work has aimed to expose compositional structure in Monte Carlo algorithms and help users explore a broad class of algorithm settings, our work aims to do the same for variational inference, exposing compositional structure in gradient estimators and variational objectives.

Formal Reasoning about Inference and Program Transformations. Our semantics builds on recent work in the denotational semantics and validation of Bayesian inference [23, 75, 76], as well as semantic foundations for differentiable programming [26, 64]. Our soundness proofs are based on logical relations [2, 26]. We also draw on a tradition of deriving probabilistic inference algorithms via program transformations [54].

10 DISCUSSION

This work gives a formal account of the automation that PPLs provide for variational inference — a powerful and widely used suite of features that has not previously been completely understood in formal programming language terms. This work's account provides a careful separation of the interactions between tracing, density computation, gradient estimation strategies, and automatic differentiation. Simultaneously, this work shows how to implement these features modularly and extensibly, addressing a number of pain points in existing implementations, and expanding the class of variational inference algorithms that users can easily express.

Limitations. That said, the modular approach we have presented has several key limitations:

- *Static checks may be unnecessarily restrictive.* For example, although the rectified linear unit (ReLU) is not differentiable at 0, in many (but not all) contexts it is safe to use it as though it were differentiable, without compromising unbiasedness. The type system of λ_{ADEV} is not sophisticated enough to distinguish when ReLU is safe or not, and so we must give it the restrictive type $\mathbb{R}^* \to \mathbb{R}^*$. Of course, at their own risk, users of the system are free to ignore these static checks.
- *No parametric discontinuities.* A key limitation of our language, shared by Pyro, ProbTorch, and Gen, is that *parametric discontinuities* (expressions that compute discontinuous functions of the input parameters) are not permitted. Variational inference is possible in these settings, and Lee et al. [36] proposed a gradient estimator that can be automated for a restricted PPL with affine discontinuities. More recently, Bangaru et al. [4] and Michel et al. [49] have presented techniques for differentiating integral expressions with parametric discontinuities. It is not yet clear to what extent the design we present could be cleanly extended to exploit these techniques.
- *User-specified variational objectives may be ill-defined.* Our correctness theorem assumes as a precondition that the objective the user has specified is well-defined (i.e., if it is defined as an expected value, that it is finite for all input parameters). Previous work has identified the verification of this condition as an important challenge for safe variational inference [35], but our system includes no static checks to do so.
- *Continuation-passing style (CPS) is unnatural in many host languages.* **adev**$\{\cdot\}$ transforms a program to CPS in order to automate unbiased gradient estimation. This is easy in our Haskell implementation, but in Python and Julia, it introduces some amount of friction. For example, in Julia, certain host-language features (like mutation, and therefore most imperative loops) are incompatible with our CPS implementation. By contrast, Pyro and Gen place few restrictions on the host-language features that can be used to define models and variational families.

In addition, our implementation does not yet incorporate several important insights and capabilities from existing deep PPLs, including Pyro's use of tensor contractions to marginalize discrete variables [55, 56], or the use of fine-grained control flow information to reduce the variance of gradient estimates [63].

Future Work. We comment on several intriguing avenues for future work:

- *The search for low variance estimators.* Our approach automates the derivation of unbiased gradient estimators, but it says little about what estimators one should choose to achieve low variance on particular problems. Our approach should make it *easier* to address this challenge, by allowing rapid exploration of a large space of estimation strategies, some of which have not been previously automated. We hope that our work and implementations might be used to carry out a study of the behavior of different gradient estimators, on a broader variety of problems than those enabled by current automation.

- *Interaction with ADEV.* Our system is the first to use ADEV [40] for scalable learning of large parameter spaces (§A.4). ADEV provides a fundamentally new perspective on automatic differentiation, and extends the technique to expected value loss functions. Our integration with ADEV has several implications: by virtue of the fact that the target language of our language's transformations is ADEV's source language, our system may be used with ADEV loss functions beyond the variational loss functions which we've discussed in this work. Extensions to ADEV which improve variance properties of gradient estimators symbiotically improve the performance of gradient estimators in our language. Indeed, we expect new investigations into low variance gradient estimators for discrete random choices [3] to open up novel variational guide families, hitherto unexplored due to poor variance or computational intractability of discrete enumeration.

DATA-AVAILABILITY STATEMENT

An artifact providing a version of `genjax.vi`, and reproducing our experiments, is available [5].

ACKNOWLEDGMENTS

The authors are grateful to Tuan Anh Le, Tan Zhi-Xuan, Cameron Freer, Andrew Bolton, George Matheos, Nishad Gothoskar, Martin Jankowiak, and Sam Witty for useful conversations and feedback, and to our anonymous referees for helpful feedback on earlier drafts of the paper. We are grateful for support from DARPA, under the DARPA Machine Common Sense (Award ID: 030523-00001) and JUMP (CoCoSys, Prime Contract No. 2023-JU-3131) programs, and DSTA, under Master Agreement No. 801899998, as well as gifts from Google, and philanthropic gifts from an anonymous donor and the Siegel Family Foundation.

REFERENCES

[1] Felix V. Agakov and David Barber. 2004. An Auxiliary Variational Method. In *Neural Information Processing (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 561–566. https://doi.org/10.1007/978-3-540-30499-9_86

[2] Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 69–83. https://doi.org/10.1007/11693024_6

[3] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. 2022. Automatic Differentiation of Programs with Discrete Randomness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/43d8e5fc816c692f342493331d5e98fc-Abstract-Conference.html

[4] Sai Praveen Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically differentiating parametric discontinuities. *ACM Trans. Graph.* 40, 4 (2021), 107:1–107:18. https://doi.org/10.1145/3450626.3459775

[5] McCoy R. Becker, Alexander K. Lew, and Xiaoyan Wang. 2024. probcomp/programmable-vi-pldi-2024: v0.1.2. Zenodo. https://doi.org/10.5281/zenodo.10935596

[6] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. , 28:1–28:6 pages. http://jmlr.org/papers/v20/18-403.html

[7] David M Blei and Michael I Jordan. 2006. Variational inference for Dirichlet process mixtures. (2006).

[8] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2016. Variational Inference: A Review for Statisticians. *CoRR* abs/1601.00670 (2016). arXiv:1601.00670 http://arxiv.org/abs/1601.00670

[9] 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.

[10] Jörg Bornschein and Yoshua Bengio. 2015. Reweighted Wake-Sleep. https://doi.org/10.48550/arXiv.1406.2751 arXiv:1406.2751 [cs].

[11] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. 2016. Importance Weighted Autoencoders. https://doi.org/10.48550/arXiv.1509.00519 arXiv:1509.00519 [cs, stat].

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

[13] Marco Cusumano-Towner and Vikash K Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/hash/acab0116c354964a558e65bdd07ff047-Abstract.html

[14] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[15] A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)* 39, 1 (1977), 1–38. https://www.jstor.org/stable/2984875 Publisher: [Royal Statistical Society, Wiley].

[16] Justin Domke. 2021. An Easy to Interpret Diagnostic for Approximate Inference: Symmetric Divergence Over Simulations. https://doi.org/10.48550/arXiv.2103.01030 arXiv:2103.01030 [cs, stat].

[17] S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. 2016. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. https://doi.org/10.48550/arXiv.1603.08575 arXiv:1603.08575 [cs].

[18] Jakob Foerster, Gregory Farquhar, Maruan Al-Shedivat, Tim Rocktäschel, Eric Xing, and Shimon Whiteson. 2018. Dice: The infinitely differentiable Monte Carlo estimator. In *International Conference on Machine Learning*. PMLR, 1529–1538.

[19] Charles W Fox and Stephen J Roberts. 2012. A tutorial on variational Bayesian inference. *Artificial intelligence review* 38 (2012), 85–95.

[20] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[21] 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.

[22] Shixiang (Shane) Gu, Zoubin Ghahramani, and Richard E Turner. 2015. Neural Adaptive Sequential Monte Carlo. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc. https://papers.nips.cc/paper_files/paper/2015/hash/99adff456950dd9629a5260c4de21858-Abstract.html

[23] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '17)*. IEEE Press, Reykjavík, Iceland, 1–12.

[24] Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. 1995. The "Wake-Sleep" Algorithm for Unsupervised Neural Networks. *Science* 268, 5214 (May 1995), 1158–1161. https://doi.org/10.1126/science.7761831

[25] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *Journal of Machine Learning Research* (2013).

[26] 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 (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

[27] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. 2021. Variational diffusion models. *Advances in Neural Information Processing Systems* 34 (2021), 21696–21707.

[28] Diederik P. Kingma, Danilo J. Rezende, Shakir Mohamed, and Max Welling. 2014. Semi-supervised learning with deep generative models. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3581–3589.

[29] Diederik P. Kingma and Max Welling. 2022. Auto-Encoding Variational Bayes. https://doi.org/10.48550/arXiv.1312.6114 arXiv:1312.6114 [cs, stat].

[30] Justin N Kreikemeyer and Philipp Andelfinger. 2023. Smoothing Methods for Automatic Differentiation Across Conditional Branches. *IEEE Access* (2023).

[31] Emile Krieken, Jakub Tomczak, and Annette Ten Teije. 2021. Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems* 34 (2021), 7574–7587.

[32] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. Automatic differentiation variational inference. *Journal of machine learning research* (2017).

[33] Tuan Anh Le, Adam R. Kosiorek, N. Siddharth, Yee Whye Teh, and Frank Wood. 2019. Revisiting Reweighted Wake-Sleep for Models with Stochastic Control Flow. , 1039–1049 pages. http://proceedings.mlr.press/v115/le20a.html

[34] Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2023. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 12:335–12:366. https://doi.org/10.1145/3571205

[35] 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 (Dec. 2019), 16:1–16:33. https://doi.org/10.1145/3371084

[36] Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization gradient for non-differentiable models. *Advances in Neural Information Processing Systems* 31 (2018).

[37] 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)*. PMLR, 1096–1106. https://proceedings.mlr.press/v180/lew22a.html

[38] 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.

[39] Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. 2023. Probabilistic Programming with Stochastic Probabilities. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 176:1708–176:1732. https://doi.org/10.1145/3591290

[40] Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 121–153. https://doi.org/10.1145/3571198 arXiv:2212.06386 [cs, stat].

[41] Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1454–1482.

[42] Michael Y. Li, Dieterich Lawson, and Scott Linderman. 2023. Neural Adaptive Smoothing via Twisting. https://openreview.net/forum?id=rC6-kGN-0v

[43] Daniel Lundén, Johannes Borgström, and David Broman. 2021. Correctness of Sequential Monte Carlo Inference for Probabilistic Programming Languages.. In *ESOP*. 404–431.

[44] Lars Maaløe, Casper Kaae Sønderby, Søren Kaae Sønderby, and Ole Winther. 2016. Auxiliary Deep Generative Models. In *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, 1445–1453. https://proceedings.mlr.press/v48/maaloe16.html ISSN: 1938-7228.

[45] Chris J. Maddison, Dieterich Lawson, George Tucker, Nicolas Heess, Mohammad Norouzi, Andriy Mnih, Arnaud Doucet, and Yee Whye Teh. 2017. Filtering Variational Objectives. https://doi.org/10.48550/arXiv.1705.09279 arXiv:1705.09279 [cs, stat].

[46] Nikolay Malkin, Salem Lahlou, Tristan Deleu, Xu Ji, Edward Hu, Katie Everett, Dinghuai Zhang, and Yoshua Bengio. 2022. GFlowNets and variational inference. *arXiv preprint arXiv:2210.00580* (2022).

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

[48] 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.

[49] Jesse Michel, Kevin Mu, Xuanda Yang, Sai Praveen Bangaru, Elias Rojas Collins, Gilbert Bernstein, Jonathan Ragan-Kelley, Michael Carbin, and Tzu-Mao Li. 2024. Distributions for Compositionally Differentiating Parametric Discontinuities. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 893–922.

[50] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. Monte Carlo gradient estimation in machine learning. *Journal of Machine Learning Research* 21, 132 (2020), 1–62.

[51] Christian Naesseth, Scott Linderman, Rajesh Ranganath, and David Blei. 2018. Variational sequential Monte Carlo. In *International conference on artificial intelligence and statistics*. PMLR, 968–977.

[52] Christian A. Naesseth, Fredrik Lindsten, and David Blei. 2020. Markovian score climbing: variational inference with KL(p||q). In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, 15499–15510.

[53] Christian A Naesseth, Fredrik Lindsten, Thomas B Schön, et al. 2019. Elements of sequential Monte Carlo. *Foundations and Trends® in Machine Learning* 12, 3 (2019), 307–392.

[54] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13*. Springer, 62–79.

[55] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P Chen. 2019. Functional tensors for probabilistic programming. *arXiv preprint arXiv:1910.10775* (2019).

[56] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019. Tensor variable elimination for plated factor graphs. In *International Conference on Machine Learning*. PMLR, 4871–4880.

[57] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. 2016. Variational autoencoder for deep learning of images, labels and captions. *Advances in Neural Information Processing Systems* 29 (2016).

[58] Alexey Radul, Adam Paszke, Roy Frostig, Matthew Johnson, and Dougal Maclaurin. 2022. You only linearize once: Tangents transpose to gradients. *arXiv preprint arXiv:2204.10923* (2022).

[59] Tom Rainforth, Adam R. Kosiorek, Tuan Anh Le, Chris J. Maddison, Maximilian Igl, Frank Wood, and Yee Whye Teh. 2018. Tighter Variational Bounds are Not Necessarily Better. https://arxiv.org/abs/1802.04537v3

[60] Rajesh Ranganath, Dustin Tran, and David Blei. 2016. Hierarchical Variational Models. In *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, 324–333. https://proceedings.mlr.press/v48/ranganath16.html ISSN: 1938-7228.

[61] D.B. Rubin. 1988. Using the SIR algorithm to simulate posterior distributions. https://api.semanticscholar.org/CorpusID:115305396

[62] Tim Salimans, Diederik Kingma, and Max Welling. 2015. Markov chain Monte Carlo and variational inference: Bridging the gap. In *International Conference on Machine Learning*. PMLR, 1218–1226.

[63] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. 2015. Gradient estimation using stochastic computation graphs. *Advances in Neural Information Processing Systems* 28 (2015).

[64] Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λS: computable semantics for differentiable programming with higher-order functions and datatypes. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 3:1–3:31. https://doi.org/10.1145/3434284

[65] Artem Sobolev and Dmitry Vetrov. 2019. Importance Weighted Hierarchical Variational Inference. https://doi.org/10.48550/arXiv.1905.03290 arXiv:1905.03290 [cs, stat].

[66] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. 2015. Learning Structured Output Representation using Deep Conditional Generative Models. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/hash/8d55a249e6baa5c06772297520da2051-Abstract.html

[67] Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning proposals for probabilistic programs with inference combinators. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. PMLR, 1056–1066. https://proceedings.mlr.press/v161/stites21a.html ISSN: 2640-3498.

[68] Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, and Alexey Radul. 2018. Simple, Distributed, and Accelerated Probabilistic Programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7609–7620. https://proceedings.neurips.cc/paper/2018/hash/201e5bacd665709851b77148e225b332-Abstract.html

[69] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Hy6b4Pqee

[70] Arash Vahdat and Jan Kautz. 2020. NVAE: A deep hierarchical variational autoencoder. *Advances in Neural Information Processing Systems* 33 (2020), 19667–19679.

[71] Dominik Wagner. 2023. *Fast and correct variational inference for probabilistic programming: Differentiability, reparameterisation and smoothing*. Ph. D. Dissertation. University of Oxford.

[72] 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.

[73] Théophane Weber, Nicolas Heess, Lars Buesing, and David Silver. 2019. Credit assignment techniques in stochastic computation graphs. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2650–2660.

[74] Heiko Zimmermann, Hao Wu, Babak Esmaeili, and Jan-Willem van de Meent. 2021. Nested Variational Inference. https://openreview.net/forum?id=kBrHzFtwdp

[75] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 83:1–83:29. https://doi.org/10.1145/3236778

[76] 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. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 60:1–60:29. https://doi.org/10.1145/3158148