Probabilistic Programming with Programmable Variational Inference

ANONYMOUS AUTHOR(S)*

Compared to the wide array of advanced Monte Carlo methods supported by modern probabilistic programming languages (PPLs), PPL support for *variational inference* is underdeveloped: users are typically limited to a small selection of predefined variational objectives and gradient estimators, which are implemented monolithically (and without explicit correctness arguments) in PPL backends. In this paper, we propose a modular approach to supporting VI in PPLs, based on compositional program transformation. First, we present a probabilistic programming language for defining models, variational families, and compositional strategies for propagating gradients. Second, we present a differentiable programming language for defining variational objectives. Models and variational families from the first language are automatically compiled into new differentiable functions that can be called from the second language, for estimating densities and expectations. Finally, we present an automatic differentiation algorithm that differentiates these variational objectives, yielding provably unbiased gradient estimators for use during optimization. We also extend our source language with features not previously supported for VI in PPLs, including approximate marginalization and normalization. This makes it possible to concisely express many models, variational families, objectives, and gradient estimators from the machine learning literature, including importance-weighted autoencoders (IWAE), hierarchical variational inference (HVI), and reweighted wake-sleep (RWS). 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 to well-established open-source PPLs.

ACM Reference Format:
Anonymous Author(s). 2023. Probabilistic Programming with Programmable Variational Inference. *J. ACM* 37, 4, Article 111 (August 2023), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Variational inference problems are optimization problems in spaces of probability distributions. Variational methods have found widespread adoption both in Bayesian settings, where the goal is to find a cheap, accurate approximation to an intractable posterior [FR12; Kuc+17; BKM17; Hof+13; BJ06], and in modern deep learning, where the goal is to fit neural, latent-variable generative models to data [KW13; Pu+16; Doe16; Kin+21; VK20; Mal+22].

Unfortunately, implementing variational inference algorithms by hand can be tedious and error-prone. The typical variational inference problem is defined by three mathematical ingredients: a family of *model distributions* $\{P_\theta \mid \theta \in \mathbb{R}^n\}$, a family of inference distributions $\{Q_\phi \mid \phi \in \mathbb{R}^m\}$ (the *variational family*), and a scalar *objective function* $\mathcal{F} \coloneqq (P, Q) \mapsto \mathcal{F}(P, Q)$ mapping particular distributions P and Q to a scalar *loss* (or *reward*). But none of these ingredients are necessarily represented directly in code. Instead, the practitioner has to:

(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. Making things worse, small changes to the objective $\mathcal{F}$, to P_θ and Q_ϕ, or to the gradient estimation strategy employed in step (1) can require large, non-local changes to M_θ and f_θ, 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 user's math, their code, or just the optimization hyperparameters.

Reflecting the importance of variational inference, many probabilistic programming languages (PPLs)—for example, Pyro [Bin+18], ProbTorch [Sti+21], and Gen [Cus+19]—feature varying degrees of automation for VI workflows. But they do so via a *menu-based* approach: users can freely specify P_θ and Q_ϕ as probabilistic programs, but then must choose from a limited set of options for the variational objective $\mathcal{F}$. Each menu option in turn supports only limited customization for the strategy used to automate a gradient estimator. This menu-based approach has several drawbacks:

- **Incomplete coverage (variational objectives).** Because each option comes packaged with a pre-set variational objective $\mathcal{F}$, many objectives that the user may wish to optimize are out of reach. For example, existing PPLs offer limited or no support for forward-KL objectives [NLB20]; hierarchical, nested, or recursive variational objectives [RTB16; Zim+21; LCM22]; symmetric divergences [Dom21]; trajectory-balance objectives [Mal+22]; SMC-based objectives [Mad+17; GGT15; LLL23; Nae+18]; and many more.
- **Incomplete coverage (gradient estimators).** Even when there is a menu option for the user's desired variational objective, it may not use the gradient estimation strategy that the user wants to apply. For example, Pyro supports the IWAE objective [BGS16], but the `RenyiELBO` option that implements it uses high-variance REINFORCE estimators to handle discrete variables in the variational family. Variance reduction strategies like data-dependent baselines or enumeration, although implemented as parts of other menu options, cannot be applied with the IWAE objective.
- **Duplicative engineering effort.** Supporting new gradient estimation strategies or language features requires separately introducing the same logic into the implementations of each menu option. Because this engineering effort is non-trivial, many capabilities are in practice not supported uniformly across the menu options. For example, as of this writing, Pyro's `ReweightedWakeSleep` objective [Le+19; BB15] does not support minibatching, even though the necessary logic has already been implemented in other particle-based objectives (such as `RenyiELBO`).
- **Difficulty of reasoning.** The monolithic implementations of each menu option intertwine various concerns, including log density accumulation, automatic differentiation, gradient estimation for random primitives, 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 [Ści+17; ŚKG18; Lew+23b; LBB21; Bor+16], pioneering work on correctness for variational inference [Lee+19; LRY23; Li+23] has generally been limited to highly restricted languages and to reasoning about specific properties (smoothness, absolute continuity), and not to end-to-end correctness of variational objective gradient estimation.

This work. In this paper, we present a highly modular, highly programmable approach to supporting variational inference in PPLs. In our approach, all three ingredients of the variational inference problem—P_θ, Q_ϕ, 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 to construct unbiased gradient estimators for the user's variational objective.

One of the keys to making this work is the recently introduced ADEV framework for modular gradient estimation [Lew+23a], which we specialize and extend to the variational inference setting. Our extensions include new constructs for differentiably tracing and estimating the densities of models and variational families (crucial for expressing virtually all variational objectives), a *reverse-mode* ADEV implementation that efficiently computes gradient estimates in one pass (instead of one pass per parameter), and the first GPU-accelerated implementation of ADEV-style AD.

Contributions. This paper contributes:

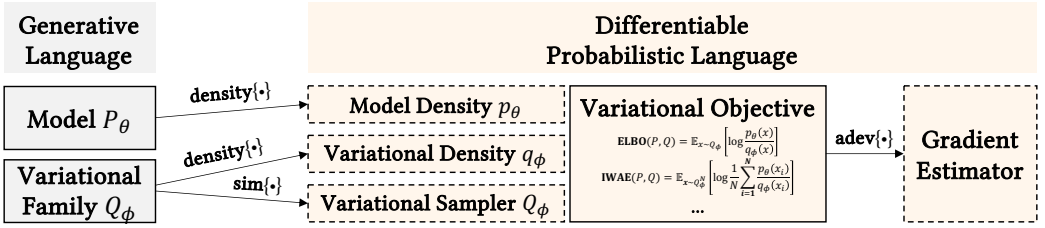


Fig. 1. We compose multiple program transformations to automate the construction of unbiased gradient estimators for variational inference. The user begins by writing programs in the generative language (gray) to encode a model and variational family. These programs are compiled into procedures for density evaluation and simulation in the probabilistic differentiable language (yellow). These automated procedures can be used to concisely define a variational objective function, e.g., the ELBO $(\theta, \phi) \mapsto \mathbb{E}_{x\sim Q_\phi}\left[\log \frac{p_\theta(x)}{q_\phi(x)}\right]$, also in the differentiable language. We can then apply the ADEV differentiation algorithm to automatically construct a *gradient estimator*, which unbiasedly estimates gradients of the user's variational objective.

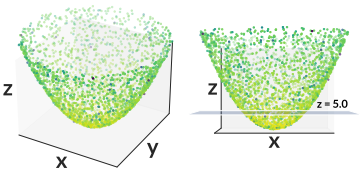
- **Modeling language**: We present an expressive language for models and variational families (§3.2), that goes beyond existing systems by supporting: (1) compositional annotation of gradient estimation strategies; and (2) first-class marginalization and normalization (§7).
- **Objective language**: In contrast to the menu-based approach of existing PPLs, we present an expressive differentiable language for variational objectives (§3.3), by extending ADEV [Lew+23a] with constructs for tracing and and estimating densities of probabilistic programs (§4).
- **Flexible, modular automation**: We automate a broad class of unbiased gradient estimators for a broad class of variational objectives. New primitive gradient estimation strategies can be added modularly with just a few lines of code, without a deep understanding of system internals.
- **Formalization**: We formalize our VI compiler 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.
- **System**: We contribute `genjax.vi`, a performant, GPU-accelerated implementation of our approach in JAX [FJL18]. 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 [Lew+23a].
- **Empirical evaluation**: We evaluate `genjax.vi` on three benchmark tasks, including inference in the challenging Attend-Infer-Repeat model [Esl+16]. Our evaluation shows that `genjax.vi` makes it possible to encode new gradient estimators that converge faster and to better solutions than Pyro's estimators. It also demonstrates, 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.[1]

2 OVERVIEW

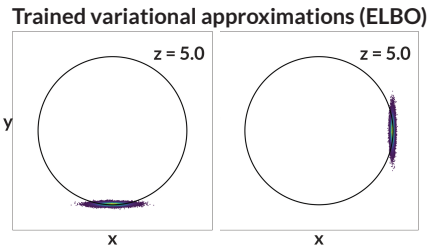
In this section, we give an overview of our approach to modular variational inference, by way of a toy example. Our approach, illustrated in Fig. 1, is based around a three-step workflow:

[1] [Lew+23a] present only a toy Haskell implementation of forward-mode ADEV, and report no experiments.

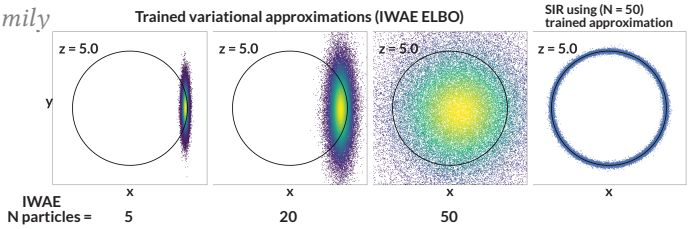
```
# Model: (x,y) latent, z observed to equal 5.0
model : G ℝ²
model = do
  x ← sample normal_REPARAM(0.0, 10.0) "x"
  y ← sample normal_REPARAM(0.0, 10.0) "y"
  let r = x² + y²
  let σ = 0.1 + (r / 100.0)
  observe normal_REPARAM(r, σ) 5.0 # z
  return (x, y)
```




```
# Variational family with parameters ϕ
naive_guide : ℝ⁴ → G ℝ²
naive_guide = λϕ. do
  let (μ₁, μ₂, log σ₁, log σ₂) = ϕ
  x ← sample normal_REPARAM(μ₁, e^(log σ₁)) "x"
  y ← sample normal_REPARAM(μ₂, e^(log σ₂)) "y"
  return (x, y)
```




```
# N-particle IS as variational family
sir_guide : ℝ⁴ → G ℝ²
sir_guide = λϕ.
  normalize model
  (importance
   (naive_guide ϕ)
   N)
```




```
# Hierarchical variational family with latent variable v
guide_joint : ℝ² → G ℝ²
guide_joint = λϕ. do
  v ← sample uniform_REPARAM(0.0, 2π) "v"
  let (log σ₁, log σ₂) = ϕ
  x ← sample normal_REPARAM(sqrt(5.0) * cos(v), e^(log σ₁)) "x"
  y ← sample normal_REPARAM(sqrt(5.0) * sin(v), e^(log σ₂)) "y"
  return (x, y)
# Marginalize the auxiliary variable with 5-particle IS
guide : ℝ² → G ℝ²
guide = λϕ.marginal ["x","y"] (guide_joint ϕ)
          (λ_.importance (sample uniform_REPARAM(0, 2π) "v") 5)
```

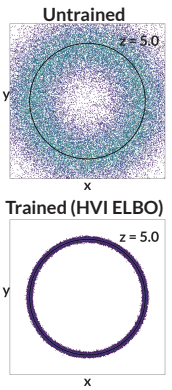


Fig. 2. An example of variational inference using our framework. (**Top**) Our model of noisy samples near a cone. (**Second**) A variational family using a pair of axis aligned normal distributions, with learnable means and standard deviations. Standard ELBO optimization using this family results in a poor fit. (**Third**) Practitioners of VI consider modified objectives derived by considering a variational family which is sampling importance resampling with a custom proposal. (**Final**) Another strategy: increasing the expressivity of the variational family by adding auxiliary latent variables.

- **Model and variational family.** The user implements a *model program* and a *variational family* (also known as a *guide program*) using the *generative language*. The generative language is a trace-based PPL that resembles Pyro [Bin+18], ProbTorch [Sti+21], and Gen [Cus+19], but goes beyond them in two ways: it features new, first-class constructs for *normalizing* and *marginalizing* probabilistic programs, and allows programs to be compositionally annotated with information about what *gradient estimation strategy* should be applied to different components.
- **Objective function.** The user's model and variational family are automatically compiled into *density estimation* and *traced simulation* procedures in a *differentiable probabilistic programming language*. The user invokes these procedures when defining a new program representing the *variational objective function* to optimize, e.g., the ELBO. Unlike in existing systems like Pyro, this objective is not chosen from a small menu of alternatives, but rather programmable by the user. The language features an expectation operator, and so standard objective functions from machine learning can be specified in just a few lines of code.
- **Derivative estimator.** The user's objective function is automatically differentiated using an extended version of the ADEV algorithm [Lew+23a], to yield an *unbiased gradient estimator* for the objective. This can then be used within stochastic gradient descent, ADAM, and similar algorithms, for optimization.

To make this concrete, consider the toy inference 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 (`model`) is based on a generative process that samples (x, y, z) noisly around a 3D cone (illustrated in the righthand panel). We use the **sample** construct to sample latent variables x and y with string-valued *names*, and the **observe** construct to condition on the observation that $z = 5.0$. The inference task is to infer x and y consistent with that observation. All primitives in our modeling language are annotated with *strategies* for propagating derivative information through them, if that need should arise; in addition to $\textbf{normal}_{\text{REPARAM}}$, our language supports $\textbf{normal}_{\text{REINFORCE}}$ and $\textbf{normal}_{\text{MVD}}$, corresponding to two other strategies.
- **Define a variational family**. In the second panel of Figure 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 `naive_guide` is a *mean-field* variational approximation, i.e., it generates x and y independently (from univariate Gaussians).
- **Define a variational objective**. We now use the differentiable programming language to define the objective function we wish to optimize. The language features three constructs that are especially useful: (1) **density**, which computes or estimates densities of probabilistic programs; (2) **sim**, which generates a pair (x, w) of a *trace* and its density from a probabilistic program; and (3) $\mathbb{E}$, which takes the expected value of a stochastic procedure. Putting them together, we can define a standard variational objective, the ELBO, or evidence lower bound:

$$\textbf{ELBO} \coloneqq \lambda\phi.\mathbb{E}(\textbf{do}\ \{(x, w_q) \leftarrow \textbf{sim}\{\texttt{naive\_guide}\}\ \phi; w_p \leftarrow \textbf{density}\{\texttt{model}\}\ x; \textbf{return}\ \log w_p - \log w_q\})$$

- **Perform stochastic optimization.** Our system automatically compiles our variational objective into an *unbiased estimator* of its *gradient* with respect to the input parameters ϕ. This estimator can be run to generate the gradient estimates required by stochastic optimization algorithms, including stochastic gradient descent and ADAM. The righthand side of Figure 2's second panel illustrates samples from the `naive_guide` after training. Because the ELBO objective is, up to a constant, the *reverse* (or *mode-seeking*) KL divergence, optimizing it yields a variational approximation that hugs one edge of the circle-shaped posterior. To better fit the whole posterior, we will need either a new objective function or a more expressive variational family.

- **Iterate on new variational families, gradient estimators, or variational objectives.** Unlike in existing PPLs that support VI, the ELBO objective is not a pre-packaged choice, and we can iterate on new objectives just by changing our program. Alternatively, we can leave it be and change the variational family. Our generative language features constructs not supported by other VI PPLs (§7), that make it easy to add expressive power to our variational family:
  - In the third panel of Fig. 2, we define `sir_guide` using the new **normalize** construct. The resulting program generates N samples from `naive_guide`, computes *importance weights* $\frac{p(x_i)}{q_\phi(x_i)}$ for each sample, and randomly selects a sample to return according to the weights. In the first three plots on the right of `sir_guide`, we show samples from `naive_guide` ϕ_N, where ϕ_N are the parameters found by optimizing the ELBO objective on `sir_guide` with N particles. As we add more particles, it becomes safer for `naive_guide` ϕ_N to spread its mass, because it will get N attempts to propose a good posterior sample. The rightmost plot illustrates samples from `sir_guide` ϕ_{50}, closely fitting the posterior. Interestingly, using **normalize** to build an importance-sampling-based variational family is equivalent to keeping `naive_guide` but changing the objective to the IWAE objective [BGS16]; both perspectives yield valid (and equivalent) implementations in our framework.
  - In the fourth panel, we define `guide` using the new **marginal** construct. First, we define `guide_joint`, which extends `guide_naive` with an *auxiliary variable* v, to allow it to better fit circle-shaped posteriors. We cannot directly use `guide_joint` with `ELBO`, however, because its traces contain `"v"`, not present in the model. The **marginal** construct shrinks the trace back down to just (x, y). The resulting algorithm is an instance of hierarchical variational inference (HVI) [RTB16].

  Note that both **marginal** and **normalize** lead to guides with intractable densities; in these cases, the weights computed by **density** and **sim** are stochastic estimates. We discuss the implications of this for VI in §7.2.

3 SYNTAX AND SEMANTICS

In this section and the following three sections, we formalize the core of our approach. Although our formal model lacks several features of our full language (discussed in detail in §7), 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. 3).

3.1 Shared Core

3.1.1 Syntax. The top of Fig. 3 presents the *shared core*, the λ-calculus on which both our probabilistic 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+23a], 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,

Shared Core

Ground types $\sigma ::= 1 \mid \mathbb{B} \mid \mathbb{I} \mid \mathbb{R}_{>0} \mid \mathbb{R}_{\geq 0} \mid \mathbb{R} \mid \mathbb{R}^* \mid \text{Str} \mid \sigma_1 \times \sigma_2$ 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 $c ::= + \mid +^* \mid < \mid \mathbf{normal}_{\text{REPARAM}} \mid \mathbf{normal}_{\text{REINFORCE}} \mid \mathbf{flip}_{\text{REINFORCE}} \mid \mathbf{flip}_{\text{ENUM}} \mid \mathbf{flip}_{\text{MVD}} \mid \ldots$

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

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

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

Generative Probabilistic Programming (λ_{Gen})

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

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

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

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

$$[\![G\,\tau]\!] = \text{Meas}^{DS}_{\ll \mathcal{B}_{\mathbb{T}}}\ \mathbb{T} \times (\mathbb{T} \Rightarrow [\![\tau]\!])$$

$$[\![\mathbf{return}\ t]\!]_1(\gamma, U) = \delta_{\{\}}(U)$$
$$[\![\mathbf{return}\ t]\!]_2(\gamma, u) = [\![t]\!](\gamma)$$
$$[\![\mathbf{sample}\ t_1\ t_2]\!]_1(\gamma, U) = \int [\![t_1]\!](\gamma, dv)\,\delta_{\{[\![t_2]\!](\gamma) \mapsto v\}}(U)$$
$$[\![\mathbf{sample}\ t_1\ t_2]\!]_2(\gamma, u) = u[[\![t_2]\!](\gamma)]$$
$$[\![\mathbf{observe}\ t_1\ t_2]\!]_1(\gamma, U) = \frac{d([\![t_1]\!](\gamma))}{d\mathcal{B}_\sigma}([\![t_2]\!](\gamma))\,\delta_{\{\}}(U)$$
$$[\![\mathbf{observe}\ t_1\ t_2]\!]_2(\gamma, u) = ()$$
$$[\![\mathbf{do}\{x \leftarrow t; m\}]\!]_1(\gamma, U) = \int [\![t]\!]_1(\gamma, du_1) \int [\![\mathbf{do}\{m\}]\!]_1(\gamma', du_2)\,\delta_{u_1 +\!\!+ u_2}(U) \cdot \mathit{disj}(u_1, u_2)$$
$$\text{where } \gamma' = \gamma[x \mapsto [\![t]\!]_2(\gamma, u_1)]$$
$$[\![\mathbf{do}\{x \leftarrow t; m\}]\!]_2(\gamma, u) = [\![\mathbf{do}\{m\}]\!](\gamma', u)$$
$$\text{where } \gamma' = \gamma[x \mapsto [\![t]\!]_2(\gamma, u)]$$

Differentiable Probabilistic Programming (λ_{ADEV})

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

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

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

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

$$[\![P\,\tau]\!] = \text{Meas}\,[\![\tau]\!] \quad | \quad [\![\widetilde{\mathbb{R}}]\!] = \text{Meas}\,\mathbb{R} \quad | \quad [\![\text{Trace}]\!] = \mathbb{T}$$

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

Fig. 3. Grammars, selected typing rules, and selected denotations for our core languages λ_{Gen} and λ_{ADEV}. smooth versions (e.g. $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$) and non-smooth versions (e.g. $+^* : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{R}^*$).[2] We allow implicit promotion of terms of type $\mathbb{R}^*$ into terms of type $\mathbb{R}$.

[2]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.

- *Primitive distributions.* The shared core includes a type $D\ \sigma$ of primitive probability distributions over ground types σ. Again following [Lew+23a], 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 unbiasedness theorem ensures that gradients will target the same objective no matter which version a program uses. 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 freely manipulate those samples. By contrast, $\mathbf{normal}_{\text{REPARAM}}$ constructs a distribution of type $D\ \mathbb{R}$, so samples 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 $[\![\Gamma]\!] := \prod_{x\in\Gamma}[\![\Gamma(x)]\!]$, 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* [Heu+17], but to ease exposition, we present our semantics mostly in terms of standard measure theory. We are careful only to do this when there is an unambiguous quasi-Borel interpretation of the measure-theoretic presentation (e.g., viewing standard Borel spaces (X, Σ_X) as their quasi-Borel counterparts (X, M_{Σ_X}) and measures μ on such spaces as their corresponding quasi-Borel measures). So, for example, we set $[\![\mathbb{R}]\!] := (\mathbb{R}, \mathcal{B}(\mathbb{R}))$ (the measurable space of reals), $[\![\sigma_1 \times \sigma_2]\!]$ to the product of the measurable spaces $[\![\sigma_1]\!]$ and $[\![\sigma_2]\!]$, and so on, but set $[\![\tau_1 \to \tau_2]\!] := [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$, 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 $\sigma \in \{1, \mathbb{B}, \text{Str}\}$, 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 $[\![D\ \sigma]\!] := \text{Prob}_{\ll\mathcal{B}_\sigma}[\![\sigma]\!]$, 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 λ_{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. 3, left).

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 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:

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

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. 3, 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. 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 $[\![t_2]\!]$, 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.[3]
- **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.

Interestingly, because **observe** statements "shave off probability mass," the semantics of a runtime error are equivalent to the semantics of observing 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 guide 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

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

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 guide.

3.3 Differentiable Probabilistic Programming with λ_{ADEV}

3.3.1 Syntax. The right panel of Fig. 3 presents a separate extension of the shared core, λ_{ADEV}, a lower-level language for differentiable probabilistic programming [Lew+23a]. 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.

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 $\exp_{\widetilde{\mathbb{R}}}$).

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 composes differently with other language constructs. 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. 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, however, 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. At a high level, this is quite simple. To simulate traces, we just run the generative program and record the value of every sample we take into a growing trace data structure. 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. Indeed, although they are not usually formalized as program transformations or rigorously proven correct, the techniques we present here are well-known and widely used in the implementations of PPLs, including for Monte Carlo.

In the context of variational inference, however, a number of thorny questions arise. 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 had to develop specialized static analyses to determine smoothness properties of different parts of programs, in order to reason

Spec

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

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

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

on types

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

on terms

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

Spec

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

Wrapper

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

Helper χ

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

on types

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

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 \{\mathbf{T},\mathbf{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}_{\text{REPARAM}}\} & := \lambda(\mu,\sigma).\mathbf{do}\{ & \chi\{\mathbf{normal}_{\text{REINFORCE}}\} & := \lambda(\mu,\sigma).\mathbf{do}\{ \\
 & \quad x \leftarrow \mathbf{normal}_{\text{REPARAM}}\ (\mu,\sigma); & & \quad x \leftarrow \mathbf{normal}_{\text{REINFORCE}}\ (\mu,\sigma); \\
 & \quad \mathbf{let}\ \rho = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}; & & \quad \mathbf{let}\ \rho = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}; \\
 & \quad \mathbf{return}(x,\rho) & & \quad \mathbf{return}(x,\rho) \\
 & \} & & \} \\
\chi\{\mathbf{flip}_{\text{ENUM}}\} & := \lambda p.\mathbf{do}\{ & \chi\{\mathbf{flip}_{\text{REINFORCE}}\} & := \lambda p.\mathbf{do}\{ \\
 & \quad b \leftarrow \mathbf{flip}_{\text{ENUM}}\ p; & & \quad b \leftarrow \mathbf{flip}_{\text{REINFORCE}}\ p; \\
 & \quad \mathbf{let}\ \rho = \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ 1-p; & & \quad \mathbf{let}\ \rho = \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ 1-p; \\
 & \quad \mathbf{return}(b,\rho) & & \quad \mathbf{return}(b,\rho) \\
 & \} & & \} \\
\chi\{\mathbf{return}\ t\} & := \mathbf{return}(\chi\{t\}, 1, \{\}) & \chi\{\mathbf{observe}\ t_1\ t_2\} & := \mathbf{do}\{\mathbf{let} w = \xi\{t_1\}(\chi\{t_2\}); \mathbf{score}\ w; \mathbf{return}\ ((), w, \{\}) \\
\chi\{\mathbf{sample}\ t_1\ t_2\} & := \mathbf{do}\{ & \chi\{\mathbf{do}\ \{x \leftarrow t; m\}\} & := \mathbf{do}\{ \\
 & \quad (x,w) \leftarrow \chi\{t_1\}; & & \quad (x,w,u_1) \leftarrow \chi\{t\}; \\
 & \quad \mathbf{return}(x,w,\{\chi\{t_2\} \mapsto x\}) & & \quad (y,v,u_2) \leftarrow \chi\{\mathbf{do}\{m\}\}; \\
 & \} & & \quad \mathbf{if}\,\mathit{disj}(u_1,u_2)\ \mathbf{then} \\
 & & & \quad\quad \mathbf{return}\ (y, wv, u_1 +\!\!+ u_2) \\
 & & & \quad \mathbf{else}\ \mathbf{do}\{\mathbf{score}\ 0; \mathbf{return}\ (y,0,\{\})\}) \\
 & & & \}
\end{array}$$

Fig. 4. Traced simulation and density evaluation as program transformations from λ_{Gen} to λ_{ADEV}.

about gradient estimation for variational inference [LRY23]. A key benefit of our approach is that it greatly simplifies this reasoning. The overall differentiability requirements for gradient estimation are enforced by the target-language (λ_{ADEV}) type system, as in [Lew+23a]. 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 (§??). That we are able to do 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.

4.1 Compiling Differentiable Density Evaluators

The density program transformation is given in Fig. 4 (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. 4). 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.

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 $\forall(\gamma, \gamma') \in R^{\xi}_{\Gamma}, ([\![\tau]\!], [\![\xi\{\tau\}]\!]) \in R^{\xi}_{\tau}$.*

The proof 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 more 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**$\{t\} : \sigma \to$ 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 in the support of μ, then $split_{\mu}$ returns that subtrace, along with

a non-empty u_2. This causes **density** to enter its **else** branch and correctly return 0 (because u is not in the support). □

4.2 Compiling Differentiable Trace Simulators

The simulation program transformation is given in Fig. 4 (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. 4 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 $\forall(\gamma, \gamma') \in R^{\chi}_{\Gamma}, ([\![\tau]\!], [\![\chi\{\tau\}]\!]) \in R^{\chi}_{\tau}$.*

Proof. By induction. □

We can then prove the correctness of **sim** itself:

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

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 ADEV programs implementing variational objectives. Once we have a λ_{ADEV} program representing our objective function, we need to differentiate it. Standard AD algorithms would not correctly handle the expectation operator $\mathbb{E}$, but the ADEV algorithm from [Lew+23a] does, to derive unbiased gradient estimators automatically. Figure 5 gives the ADEV program transformation, extended to handle new datatypes (traces) and unnormalized measures (due to **score**). Figure 5 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:[4]

Definition 5.1 (locally dominated). A function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is locally domianted 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 $\forall \theta' \in U(\theta), \forall x \in \mathbb{R}, |f(\theta', x)| \leq m_{U(\theta)}(x)$.

[4]We have omitted several terms from Figure 5, denoted with "...". These terms are as in [Lew+23a, Fig. 26], and do not affect the behavior of the **adev** transformation, only **dom**.

Spec

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

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

Wrapper

$\mathbf{adev}\{t\} \coloneqq \lambda\theta.\lambda\mathbf{v}.\mathbb{E}(\mathbf{do}\{(y, y') \leftarrow \mathcal{D}\{t\}\ ((\theta_1, \mathbf{v}_1), \ldots, (\theta_n, \mathbf{v_n})); \mathbf{return}\ y'\})$

$\mathbf{dom}\{t\}_{(\theta,i)} \coloneqq \lambda(\phi, x).\pi_2(\pi_2(\mathcal{D}\{t\}((\theta_1, 0), \ldots, (\theta_{i-1}, 0), (\phi, 1), (\theta_{i+1}, 0), \ldots, (\theta_n, 0))\ x$

Helper $\mathcal{D}$

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

on types

$\mathcal{D}\{\mathbb{R}\} \coloneqq \mathbb{R} \times \mathbb{R}$

$\mathcal{D}\{\sigma\} \coloneqq \sigma \quad (\sigma \in \{\mathbb{R}^*, \mathbb{B}, \text{Str}\})$

$\mathcal{D}\{\tau_1 \times \tau_2\} \coloneqq \mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\}$

$\mathcal{D}\{\tau_1 \to \tau_2\} \coloneqq \mathcal{D}\{\tau_1\} \to \mathcal{D}\{\tau_2\}$

$\mathcal{D}\{\text{Trace}\} \coloneqq \text{Trace}$

$\mathcal{D}\{D\ \sigma\} \coloneqq \mathcal{D}\{P\ \sigma\}$

$\mathcal{D}\{\widetilde{\mathbb{R}}\} \coloneqq P\ (\mathbb{R} \times \mathbb{R}) \times (\mathbb{R}^* \to \mathbb{R} \times \mathbb{R})$

$\mathcal{D}\{P\ \tau\} \coloneqq (\mathcal{D}\{\tau\} \to \mathcal{D}\{\widetilde{\mathbb{R}}\}) \to \mathcal{D}\{\widetilde{\mathbb{R}}\}$

$R^{\mathcal{D}}_{\mathbb{R}} \coloneqq \{(f, g) \mid g = \lambda r.(f(r), f'(r))\}$

$R^{\mathcal{D}}_{\sigma} \coloneqq \{(f, f) \mid f \text{ constant}\}$

$R^{\mathcal{D}}_{\tau_1\times\tau_2} \coloneqq \{(f, g) \mid \forall i \in \{1, 2\}.(\pi_i \circ f, \pi_i \circ g) \in R^{\mathcal{D}}_{\tau_i}\}$

$R^{\mathcal{D}}_{\tau_1\to\tau_2} \coloneqq \{(f, g) \mid \forall (x, y) \in R^{\mathcal{D}}_{\tau_1}, (\lambda r.f(r)(x(r)), g(r)(y(r))) \in R^{\mathcal{D}}_{\tau_2}\}$

$R^{\mathcal{D}}_{\text{Trace}} \coloneqq \{(f, g) \mid \forall k \in [\![\text{Str}]\!].(\lambda r.f(r)[k], \lambda r.g(r)[k]) \in R^{\mathcal{D}}_{\sigma}\}$

$R^{\mathcal{D}}_{D\ \sigma} \coloneqq R^{\mathcal{D}}_{P\ \sigma}$

$R^{\mathcal{D}}_{\widetilde{\mathbb{R}}} \coloneqq \{(\mu, \nu) \mid \forall \theta.\ \int_{\mathbb{R}} h_1(\theta)(s)ds = \int_{\mathbb{R}} x\mu(\theta, dx) = \mathbb{E}_{x\sim\pi_{1*}(\pi_1\circ\nu)(\theta)}[x] \wedge \forall \theta.\ \int_{\mathbb{R}} h_2(\theta)(s)ds = \mathbb{E}_{x\sim\pi_{2*}(\pi_{1*}\nu)(\theta)}[x] \wedge (\lambda\theta.\lambda s.h_1(\theta)(s), \lambda\theta.\lambda s.(h_1(\theta)(s), h_2(\theta)(s))) \in R_{\mathbb{R}^*\to\mathbb{R}}\}$ where $h_i \coloneqq \lambda\theta.\lambda s.\pi_i((\pi_2 \circ \nu)(\theta)(s))$

$R^{\mathcal{D}}_{P\ \tau} \coloneqq \{(\mu, \nu) \mid (\lambda r.\lambda k.\lambda U.\mathbb{E}_{x\sim\mu(r)}[k(x, U)], \nu) \in R^{\mathcal{D}}_{(\tau\to\widetilde{\mathbb{R}})\to\widetilde{\mathbb{R}}}\}$

on terms

$\mathcal{D}\{()\} \coloneqq ()$

$\mathcal{D}\{c\} \coloneqq c_{\mathcal{D}}$

$\mathcal{D}\{\lambda x.t\} \coloneqq \lambda x.\mathcal{D}\{t\}$

$\mathcal{D}\{(t_1, t_2)\} \coloneqq (\mathcal{D}\{t_1\}, \mathcal{D}\{t_2\})$

$\mathcal{D}\{b\}\ (b \in \{\mathbf{T}, \mathbf{F}\}) \coloneqq b$

$\mathcal{D}\{\{t_1 \mapsto t_2\}\} \coloneqq \{\mathcal{D}\{t_1\} \mapsto \mathcal{D}\{t_2\}\}$

$\mathcal{D}\{t_1[t_2]\} \coloneqq \mathcal{D}\{t_1\}[\mathcal{D}\{t_2\}]$

$\mathcal{D}\{r\} \coloneqq (r, 0)$

$\mathcal{D}\{\mathbf{return}\ t\} \coloneqq \lambda\kappa.\kappa(\mathcal{D}\{t\})$

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

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

$\mathcal{D}\{\{\}\} \coloneqq \{\}$

$\mathcal{D}\{x\} \coloneqq x$

$\mathcal{D}\{t_1\ t_2\} \coloneqq \mathcal{D}\{t_1\}\ \mathcal{D}\{t_2\}$

$\mathcal{D}\{\pi_i\ t\} \coloneqq \pi_i\ \mathcal{D}\{t\}$

$\mathcal{D}\{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\} \coloneqq \mathbf{if}\ \mathcal{D}\{t\}\ \mathbf{then}\ \mathcal{D}\{t_1\}\ \mathbf{else}\ \mathcal{D}\{t_2\}$

$\mathcal{D}\{t_1 + t_2\} \coloneqq \mathcal{D}\{t_1\} + \mathcal{D}\{t_2\}$

$\mathcal{D}\{\mathbf{sample}\ t\} \coloneqq \mathcal{D}\{t\}$

$\mathcal{D}\{\mathbf{score}\ t\} \coloneqq \lambda\kappa.(\mathbf{do}\{y \leftarrow \pi_1(\kappa(())); \mathbf{return}(\mathcal{D}\{t\} \times_{\mathcal{D}} y)\}, \lambda s.(\pi_2(\kappa(()))\ s) \times_{\mathcal{D}} \mathcal{D}\{t\})$

$\mathcal{D}\{\mathbf{do}\ \{x \leftarrow t; m\}\} \coloneqq \lambda\kappa.\mathcal{D}\{t\}(\lambda x.\mathcal{D}\{\mathbf{do}\{m\}\}(\kappa))$

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

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

Fig. 5. Monte Carlo gradient estimation as a program transformation from λ_{ADEV} to λ_{ADEV}.

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 Figure 5 gives the logical relations. It extends the proof of [Lew+23a] with new cases, for **score**, as well as for trace construction and manipulation.

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:

Theorem 6.1. *Let:*

- $\vdash P : \mathbb{R}^n \to G\ \tau_1$, *the model program, be a closed* λ_{Gen} *term;*
- $\vdash Q : \mathbb{R}^m \to G\ \tau_2$, *the variational program, be a closed* λ_{Gen} *term;*
- Γ *be the* λ_{ADEV} *environment* $p : Trace \to \mathbb{R}, q : Trace \to \mathbb{R}, \mathbb{P} : P(Trace \times \mathbb{R}), \mathbb{Q} : P(Trace \times \mathbb{R})$;
- $\Gamma \vdash t : \widetilde{\mathbb{R}}$, *the objective program, be an open* λ_{ADEV} *term in context* Γ;
- $\mathcal{F}(\mu_P, \mu_Q)$, *the objective function, equal* $\int_{\mathbb{R}} x\, [\![t]\!]\left(\frac{d\mu_P}{d\mathcal{B}_{\mathbb{T}}}, \frac{d\mu_Q}{d\mathcal{B}_{\mathbb{T}}}, (id \otimes \frac{d\mu_P}{d\mathcal{B}_{\mathbb{T}}})_* \mu_P, (id \otimes \frac{d\mu_Q}{d\mathcal{B}_{\mathbb{T}}})_* \mu_Q, dx\right)$;
- $\mathcal{L}(\theta, \phi)$, *the parametric objective function, equal* $\mathcal{F}([\![P]\!]_1(\theta), [\![Q]\!]_1(\phi))$;
- $s \coloneqq \vdash \lambda(\theta, \phi).t[\boldsymbol{density}\{P\}\ \theta/p, \boldsymbol{density}\{Q\}\ \phi/q, \boldsymbol{sim}\{P\}\ \theta/\mathbb{P}, \boldsymbol{sim}\{Q\}\ \phi/\mathbb{Q}] : \mathbb{R}^{n+m} \to \widetilde{\mathbb{R}}$ *be the compiled objective, a closed term of* λ_{ADEV}.

Further assume that for every $\theta \in \mathbb{R}^n$ *and* $\phi \in \mathbb{R}^m$, $\mathcal{L}(\theta, \phi)$ *is finite, and* $[\![\boldsymbol{dom}\{s\}_{((\theta,\phi),i)}]\!]$ *is locally dominated for every* $i \in \underline{n + m}$. *Then* $[\![\mathbf{adev}\{s\}]\!]((\theta, \phi), \mathbf{v})$ *is a probability measure with finite expectation for all* $\theta \in \mathbb{R}^n$, $\phi \in \mathbb{R}^m$, *and* $v \in \mathbb{R}^{n+m}$, *and* $\mathbb{E}_{x \sim [\![\mathbf{adev}\{s\}]\!]((\theta,\phi),\mathbf{v})}[x] = (\nabla_{(\theta,\phi)} \mathcal{L}(\theta, \phi))^T \mathbf{v}$.

7 FULL LANGUAGE

In the previous sections, we presented a formal model of the key features of our approach. We now elaborate on our full language and system, which extends our formal model in three key ways:

- **New language features for probabilistic models and variational families (§7.1).** Our full language includes constructs for *marginalizing* and *normalizing* probabilistic programs, making it possible to express a broader class of variational families than in current systems.
- **Differentiable stochastic estimators of densities and density reciprocals (§7.2).** Our full language can compile differentiable unbiased estimators of density functions, when exact densities cannot be efficiently computed. These estimators can even have learnable parameters, which can be optimized jointly as part of the overall variational objective.
- **Reverse-mode automatic differentiation of expected values (Appendix A).** 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.

The remainder of this section introduces these features, explains their implications for programmable variational inference, and highlights the key innovations necessary to implement them.

7.1 New Language Features for Expressive Models and Variational Families

The full language for models and variational families modularly extends our formalized core with two new constructs: **marginal**, for marginalizing auxiliary variables from the model or variational family; and **normalize**, for building variational families that use Monte Carlo to approximate the normalized posteriors of other programs. We discuss each feature in turn.

Marginalization of auxiliary variables: A key requirement in variational inference is that the model and variational family be defined over the same set of random variables [Lee+19; Lew+19; WHR21; Li+23]. But it is often desirable to introduce *auxiliary variables* to only the variational family, or only the model, in order to make the *marginal* distribution on the shared variables more interesting. Figure 2 illustrates one example. Both the model and variational family are defined over x and y, and intuitively, we want the variational family to learn a distribution that concentrates mass around the

circumference of a circle of radius $\sqrt{5}$. But using Gaussian primitives alone we can never express such a distribution. Therefore, `guide_joint` introduces an auxiliary variable $v \sim U([0, 2\pi])$, which informs the location of both x and y. Because `guide_joint` is now defined over the three variables (v, x, y), it is no longer directly usable as a variational family for the model. Using **marginal**, we can fix the problem, constructing a new program representing `guide_joint`'s marginal on x and y.

The syntax for marginalizing is **marginal** *names program algorithm*. The argument *names* is a list of strings indicating which of the variables from *program* should be kept. Marginal distributions may have intractable density functions, and so our system automates *differentiable stochastic estimates* of their densities instead (§7.2). The argument *algorithm* specifies a strategy for calculating these estimates. It is given as a function, taking in values for the variables in *names*, and outputting an *algorithm* for inferring the posterior over auxiliary variables (see §7.2 for details).

With **marginal**, users can encode within a PPL many variational inference algorithms that cannot be easily expressed in existing systems. For example, hierarchical variational inference (HVI) [RTB16; AB04] can be implemented by introducing then marginalizing auxiliary variables in a variational family, using 1-particle importance sampling as the *algorithm* argument. Increasing the number of particles used to estimate the marginal density of the variational family yields importance-weighted HVI [SV19]. Parameterizing the proposal used to marginalize the variational family as a neural network recovers auxiliary deep generative models (ADGM) [Maa+16]. Nesting a **marginal** distribution as the proposal to marginalize a variational family is an instance of recursive auxiliary-variable inference (RAVI) [LCM22]. Extending the variational family with steps from a Markov chain Monte Carlo algorithm, then marginalizing those, implements Markov chain variational inference (MCVI) [SKW15]. And many aspects of these algorithms could in principle be composed to develop new inference algorithms altogether.

Approximate normalization: Another way to make variational families more expressive is to add general-purpose inference logic to the variational family itself. To do this, users can apply our **normalize** construct, which builds a probabilistic program that represents the output distribution of a given inference algorithm applied to a given (unnormalized) probabilistic program. Its syntax is **normalize** *program algorithm*. Figure 2 illustrates a typical usage: after defining a simple variational family (`naive_guide`), we define an improved variational family (`sir_guide`) that performs N-particle importance sampling, targeting the model's posterior, using `naive_guide` as a proposal distribution. This normalized variational family can then be used in several ways. If we use `sir_guide` as a variational family and optimize the usual ELBO objective (using our stochastically estimated densities for `sir_guide`, see §7.2), this is equivalent to optimizing the IWAE (or IWELBO) objective with `naive_guide` as the variational family [BGS16]. Alternatively, we can optimize the objective $\phi \mapsto \mathbb{E}_{x \sim [\![\texttt{sir\_guide}]\!]_{\phi_0}}[-\log [\![\texttt{naive\_guide}]\!]_\phi(x)]$, which tunes `naive_guide` to minimize its KL divergence to the approximate inference algorithm `sir_guide`. This latter objective is the wake-phase ϕ objective in the reweighted wake sleep algorithm [BB15; Le+19].

7.2 Differentiable Stochastic Estimators of Densities and their Reciprocals

With the new features from §7.1, it is possible to define models and variational families whose exact densities cannot be efficiently computed. As such, in our full language, **density** and **sim** do not produce exact density evaluators, but rather density estimators that may be stochastic, depending on the input program. Importantly, these estimators are still implemented in the ADEV language, still come equipped with gradient estimation strategies, and are still guaranteed to satisfy the necessary differentiability requirements for gradients to be unbiased. We now discuss the implications of this stochastic estimation for users of our language; see Appendix C an overview of how the stochastic estimates are computed.

For users: On the surface, very little changes when densities are made stochastic. The automated **density** procedures now have type Trace $\to P\ \mathbb{R}$, instead of Trace $\to \mathbb{R}$, and the specifications for both **density** and **sim** change slightly: now, **density**$\{t\}$ produces only *unbiased estimates* of the density of $[\![t]\!]_1$, and **sim**$\{t\}$ produces pairs (x, w) with the property that $\mathbb{E}[\frac{1}{w} \mid x]$ is the reciprocal density of $[\![t]\!]_1$ at x. But using density estimators instead of exact densities can change the meaning of user-specified variational objectives. Consider, for example, the ELBO, whose implementation is very slightly modified to account for the new type of **density**:

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

This program may no longer denote the usual ELBO objective, $\int \log \frac{\tilde{p}_\theta(x)}{q_\phi(x)}\ Q_\phi(x)dx$. Instead, it denotes the objective

$$(\theta, \phi) \mapsto \iint \log \frac{w_p}{w_q} \kappa_{p;\theta}(x, dw_p)\kappa_{q;\phi}(dx, dw_q),$$

where we write κ_p for the unbiased density estimation kernel for P_θ (the new specification for **density**$\{t_p\}$) and κ_q for the 'reciprocal density simulation' measure for Q_ϕ (the new specification for **sim**$\{t_q\}$). Fortunately, however, this is still a sensible variational objective. Because $\mathbb{E}[w_p \mid x] = \tilde{p}_\theta(x)$ and $\mathbb{E}[\frac{1}{w_q} \mid x] = \frac{1}{q_\phi(x)}$, we can apply Jensen's inequality and obtain that $\mathbb{E}[\log w_p \mid x] \leq \log \tilde{p}_\theta(x)$ and $\mathbb{E}[\log \frac{1}{w_q} \mid x] \leq \log \frac{1}{q_\phi(x)}$. This in turn implies that our new objective is a *lower bound* on the original ELBO. In fact, it can be shown that the new bound decomposes as the original ELBO plus a non-negative term related to the average variation of the stochastic density estimates [LCM22]. Thus, optimizing this modified ELBO simultaneously maximizes the original ELBO and minimizes the variability of the density estimates.

8 EVALUATION

We evaluate our approach using a prototype implementation of our language (`genjax.vi`), implemented as an extension to the Gen [Cus+19] probabilistic programming language (in JAX). We consider several case studies designed to evaluate the following criteria:

- (**Overhead**) *Is there a computational cost to our automation compared to hand-coded estimators?* We compare the same gradient estimator for a variational autoencoder [KW22] 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 [Esl+16] and investigate the capabilities of our system compared to Pyro [Bin+18] - an industry supported system with a focus on variational inference as a core capability.
- (**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 [AB04].

Device characteristics All experiments were run on a device with an AMD Ryzen 7 7800X3D @ 5.050 GHz and a Nvidia RTX 4090.

Overhead In Table 1, we illustrate a comparison between our implementation, and a manually coded implementation of the ELBO gradients (in JAX, using `jax.grad` and `tensorflow.probability` distributions - c.f. Appendix B). We consider comparisons for minibatch gradient computation time, as well as epoch time (the time it takes to sweep through the entire MNIST dataset, with minibatch sizes of $n = 64$). For minibatch gradient comparisons, we find that our automation introduces a small amount of runtime overhead (on order of 3-10%) compared to our hand coded

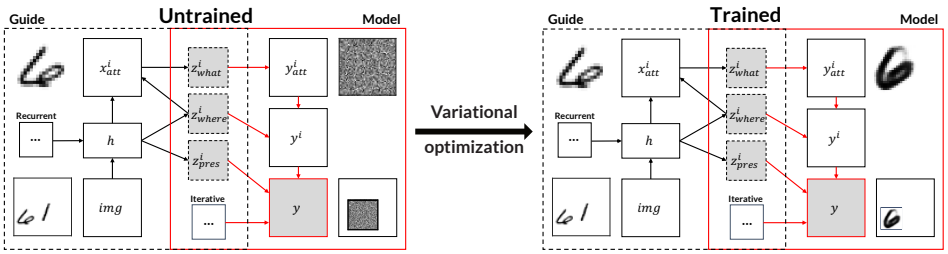


Fig. 6. AIR is a generative model for multi-object images, which is trained variationally. The generative model randomly selects a number (maximum 3) of patches to render onto a canvas, selecting a location, scale, and latent code for each. The variational family predicts these latent variables from an image. The model is trained on a dataset of images constructed by randomly translating and scaling 3 or fewer MNIST digits onto a canvas.

Table 2. Benchmark timings (in seconds) for a single training epoch using mini-batches of size 64 for the AIR model and variational family [Esl+16] on MultiMNIST, for ELBO and IWELBO (IWAE) objectives. Each estimator name denotes a run where all the discrete random choices in AIR use the corresponding estimator. IWAE indicates runs where the IWELBO objective with $n = 2$ particles was used.

System	Compiler	REINFORCE	ENUM	MVD	IWAE + REINFORCE	IWAE + 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

implementation (shown in JAX, in Fig. 10 in Appendix). In full epoch computations, where we compute and apply gradients to the parameters of our model and guide in a tight loop (about ~940 iterations), the overhead compounds leading to a runtime penalty on the order of ~ 1ms (Figure 10, top right, bottom). Our program transformations also induce a compile time penalty - but we found this penalty to be negligible and not statistically significant for the VAE program. In other implementations of our language, we'd expect the runtime cost associated with CPS to incur a penalty: our JAX implementation negates this by performing partial evaluation to remove continuations at compile time.

Overall performance We consider *Attend, Infer, Repeat* [Esl+16] (AIR) - a deep generative model of multi-object images. We compare convergence in accuracy and objective value against wall clock time in Figure 7 for several estimators expressed in our system and in Pyro. In our results, estimator names refer to the strategies used to treat the discrete random choices in AIR. Micro-benchmark timing results are shown in Table 2. Our implementation's performance is highly competitive, and we compositionally support a broader class of estimators and objectives than Pyro - e.g. allowing measure valued derivative estimators for Bernoulli distributions as a gradient estimator strategy, for the IWAE objective. We can easily swap objectives (as programs in our loss language), e.g. using Reweighted Wake-Sleep (RWS) instead of the ELBO (or IWAE instead of ELBO) without compromising the correctness of our gradient estimators. We use these capabilities to construct novel estimators for AIR (e.g., MVD for IWAE) that converge faster than standard estimators.

Table 1. Mini-batch gradient benchmark comparing our implementation versus hand coded estimators for standard VAE. Time in ms.

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 4. Mean objective value (in nats) on repeated runs for several variational objectives, including ones which utilize **marginal**.

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

Expressivity and compositional correctness In Table 3, we enumerate a subset of possible combinations of gradient estimation strategies and loss objectives for the AIR model supported by our system. In Table 4, we consider the model shown in Figure 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 (estimates of lower bounds of $\log p(x)$) for different variational objectives. While ELBO and IWELBO are standard, our system allows using more expressive approximations and tight lower bound objectives *compositionally* (such as DIWHVI [SV19], which uses SIR with particle number m internally to perform marginalization over auxiliary latent variables, in the IWAE objective with outer particle number n) to achieve tighter bounds.

Grad. Estimation Strategies RE.	EN.	BL.	MV.	Objective	Batch	System 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	×	✓

Table 3. We consider a combinatorial space of gradient estimators and objective functions that can be optimized for the Attend-Infer-Repeat model. Pyro supports only a pre-selected menu of options, whereas `genjax.vi` can express them all, compositionally.

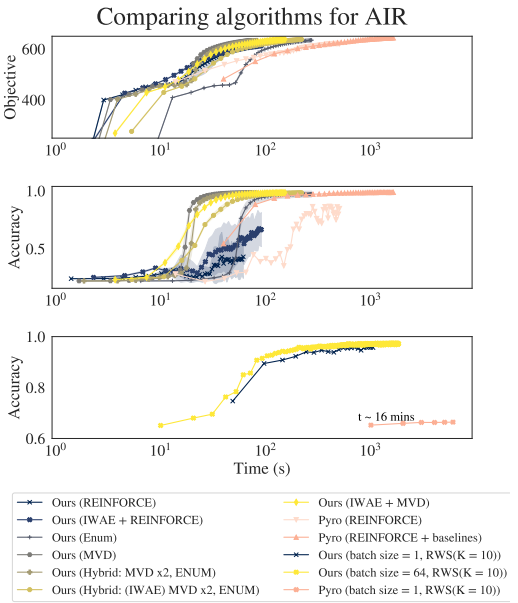


Fig. 7. We evaluate a variety of custom estimators and objectives (ELBO, IWAE, RWS) using our system. Our on average best estimator (IWAE + MVD, not expressible in Pyro) converges an order of magnitude faster than Pyro's recommended estimator.

9 RELATED WORK

Variational inference in PPLs. Many PPLs support some form of variational inference [Bin+18; Sti+21; Car+17; GXG18; Cus+19], and it is the primary focus of Pyro [Bin+18] and ProbTorch [Sti+21].

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 [Zim+21; Sti+21]. Pyro supports by far the most gradient estimators and objectives of existing variational PPLs and has extensive documentation. Pyro also supports some variance reduction strategies that we have not yet implemented, e.g. exploiting conditional independence using their **plate** operator. However, extending Pyro with new variational objectives or gradient estimation strategies requires deeply understanding (and interacting with) its internals. Furthermore, Pyro's modeling language does not have our constructs for marginalization and normailization (although Pyro has some support for marginalizing discrete, finite-support auxiliary variables from models).

Within the PL community, researchers have made some progress toward formalizing and developing static analyses for ensuring soundness properties of variational inference [Lee+19; LRY23; WHR21], as well as program transformations for automatically constructing variational families informed by the model's structure [Li+23]. 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 [LRY23]'s analysis to automatically annotate unannotated λ_{Gen} programs with gradient estimators.

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 [KTT21; Sch+15; Web+19], imperative programs with discrete randomness [Ary+23], and higher-order probabilistic programs [Lew+23a]. 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 [Lew+23a]. 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 formalized for programs with vector and not just scalar inputs; 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 [MSP14; Man+18; Nar+16; Cus+19; Lew+23b]. In much the same ways that this prior work has aimed to make Monte Carlo inference more programmable and less "menu-based", 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 [Ści+17; Heu+17; ŚKG18], as well as semantic foundations for differentiable programming [HSV20; SMC21]. Our soundness proofs are based on logical relations [Ahm06; HSV20]. We also draw on a tradition of deriving probabilistic inference algorithms via program transformations [Nar+16].

REFERENCES

[RR88] Db Rubin and Db Rubin. "Using the SIR algorithm to simulate posterior distributions". In: 1988. URL: https://api.semanticscholar.org/CorpusID:115305396.

[AB04] Felix V. Agakov and David Barber. "An Auxiliary Variational Method". en. In: *Neural Information Processing*. Ed. by Nikhil Ranjan Pal et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 561–566. ISBN: 978-3-540-30499-9. DOI: 10.1007/978-3-540-30499-9_86.

[Ahm06] Amal Ahmed. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". en. In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 69–83. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_6.

[BJ06] David M Blei and Michael I Jordan. "Variational inference for Dirichlet process mixtures". In: (2006).

[FR12] Charles W Fox and Stephen J Roberts. "A tutorial on variational Bayesian inference". In: *Artificial intelligence review* 38 (2012), pp. 85–95.

[Hof+13] Matthew D Hoffman et al. "Stochastic variational inference". In: *Journal of Machine Learning Research* (2013).

[KW13] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[MSP14] Vikash Mansinghka, Daniel Selsam, and Yura Perov. "Venture: a higher-order probabilistic programming platform with programmable inference". In: *arXiv preprint arXiv:1404.0099* (2014).

[BB15] Jörg Bornschein and Yoshua Bengio. *Reweighted Wake-Sleep*. arXiv:1406.2751 [cs]. Apr. 2015. DOI: 10.48550/arXiv.1406.2751. URL: http://arxiv.org/abs/1406.2751 (visited on 09/07/2023).

[GGT15] Shixiang (Shane) Gu, Zoubin Ghahramani, and Richard E Turner. "Neural Adaptive Sequential Monte Carlo". In: *Advances in Neural Information Processing Systems*. Vol. 28. Curran Associates, Inc., 2015. URL: https://papers.nips.cc/paper_files/paper/2015/hash/99adff456950dd9629a5260c4de21858-Abstract.html (visited on 09/23/2023).

[SKW15] Tim Salimans, Diederik Kingma, and Max Welling. "Markov chain monte carlo and variational inference: Bridging the gap". In: *International conference on machine learning*. PMLR. 2015, pp. 1218–1226.

[Sch+15] John Schulman et al. "Gradient estimation using stochastic computation graphs". In: *Advances in neural information processing systems* 28 (2015).

[Bor+16] Johannes Borgström et al. "A lambda-calculus foundation for universal probabilistic programming". In: *ACM SIGPLAN Notices* 51.9 (2016), pp. 33–46.

[BGS16] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. *Importance Weighted Autoencoders*. arXiv:1509.00519 [cs, stat]. Nov. 2016. DOI: 10.48550/arXiv.1509.00519. URL: http://arxiv.org/abs/1509.00519 (visited on 09/07/2023).

[Doe16] Carl Doersch. "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908* (2016).

[Esl+16] S. M. Ali Eslami et al. *Attend, Infer, Repeat: Fast Scene Understanding with Generative Models*. arXiv:1603.08575 [cs]. Aug. 2016. DOI: 10.48550/arXiv.1603.08575. URL: http://arxiv.org/abs/1603.08575 (visited on 10/09/2023).

[Maa+16] Lars Maaløe et al. "Auxiliary Deep Generative Models". en. In: *Proceedings of The 33rd International Conference on Machine Learning*. ISSN: 1938-7228. PMLR, June 2016, pp. 1445–1453. URL: https://proceedings.mlr.press/v48/maaloe16.html (visited on 10/23/2023).

[Nar+16] Praveen Narayanan et al. "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. 2016, pp. 62–79.

[Pu+16] Yunchen Pu et al. "Variational autoencoder for deep learning of images, labels and captions". In: *Advances in neural information processing systems* 29 (2016).

[RTB16] Rajesh Ranganath, Dustin Tran, and David Blei. "Hierarchical Variational Models". en. In: *Proceedings of The 33rd International Conference on Machine Learning*. ISSN: 1938-7228. PMLR, June 2016, pp. 324–333. URL: https://proceedings.mlr.press/v48/ranganath16.html (visited on 10/13/2023).

[BKM17] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. "Variational inference: A review for statisticians". In: *Journal of the American statistical Association* 112.518 (2017), pp. 859–877.

[Car+17] Bob Carpenter et al. "Stan: A probabilistic programming language". In: *Journal of statistical software* 76 (2017).

[CM17] Marco Cusumano-Towner and Vikash K Mansinghka. "AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/acab0116c354964a558e65bdd07ff047-Abstract.html (visited on 09/23/2023).

[Heu+17] Chris Heunen et al. "A convenient category for higher-order probability theory". In: *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '17. Reykjavík, Iceland: IEEE Press, June 2017, pp. 1–12. ISBN: 978-1-5090-3018-7. (Visited on 11/10/2023).

[Kuc+17] Alp Kucukelbir et al. "Automatic differentiation variational inference". In: *Journal of machine learning research* (2017).

[Mad+17] Chris J. Maddison et al. *Filtering Variational Objectives*. arXiv:1705.09279 [cs, stat]. Nov. 2017. DOI: 10.48550/arXiv.1705.09279. URL: http://arxiv.org/abs/1705.09279 (visited on 10/09/2023).

[Ści+17] Adam Ścibior et al. "Denotational validation of higher-order Bayesian inference". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), 60:1–60:29. DOI: 10.1145/3158148. URL: https://dl.acm.org/doi/10.1145/3158148 (visited on 11/10/2023).

[Bin+18] Eli Bingham et al. *Pyro: Deep Universal Probabilistic Programming*. arXiv:1810.09538 [cs, stat]. Oct. 2018. DOI: 10.48550/arXiv.1810.09538. URL: http://arxiv.org/abs/1810.09538 (visited on 10/11/2023).

[Foe+18] Jakob Foerster et al. "Dice: The infinitely differentiable monte carlo estimator". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1529–1538.

[FJL18] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing". In: *Systems for Machine Learning* 4.9 (2018).

[GXG18] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: a language for flexible probabilistic inference". In: *International conference on artificial intelligence and statistics*. PMLR. 2018, pp. 1682–1690.

[Man+18] Vikash K Mansinghka et al. "Probabilistic programming with programmable inference". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 603–616.

[Nae+18] Christian Naesseth et al. "Variational sequential monte carlo". In: *International conference on artificial intelligence and statistics*. PMLR. 2018, pp. 968–977.

[ŚKG18] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. "Functional programming for modular Bayesian inference". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (July 2018), 83:1–83:29. DOI: 10.1145/3236778. URL: https://dl.acm.org/doi/10.1145/3236778 (visited on 11/10/2023).

[Cus+19] Marco F. Cusumano-Towner et al. "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. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 221–236. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314642. URL: https://dl.acm.org/doi/10.1145/3314221.3314642 (visited on 10/14/2023).

[Le+19] Tuan Anh Le et al. *Revisiting Reweighted Wake-Sleep for Models with Stochastic Control Flow*. arXiv:1805.10469 [cs, stat]. Sept. 2019. DOI: 10.48550/arXiv.1805.10469. URL: http://arxiv.org/abs/1805.10469 (visited on 10/23/2023).

[Lee+19] Wonyeol Lee et al. "Towards verified stochastic variational inference for probabilistic programs". In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), 16:1–16:33. DOI: 10.1145/3371084. URL: https://dl.acm.org/doi/10.1145/3371084 (visited on 11/10/2023).

[Lew+19] Alexander K Lew et al. "Trace types and denotational semantics for sound programmable inference in probabilistic languages". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32.

[NLS+19] Christian A Naesseth, Fredrik Lindsten, Thomas B Schön, et al. "Elements of sequential monte carlo". In: *Foundations and Trends® in Machine Learning* 12.3 (2019), pp. 307–392.

[SV19] Artem Sobolev and Dmitry Vetrov. *Importance Weighted Hierarchical Variational Inference*. arXiv:1905.03290 [cs, stat]. May 2019. DOI: 10.48550/arXiv.1905.03290. URL: http://arxiv.org/abs/1905.03290 (visited on 10/23/2023).

[Web+19] Théophane Weber et al. "Credit assignment techniques in stochastic computation graphs". In: *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR. 2019, pp. 2650–2660.

[HSV20] Mathieu Huot, Sam Staton, and Matthijs Vákár. "Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing". en. In: *Foundations of Software Science and Computation Structures*. Ed. by Jean Goubault-Larrecq and Barbara König. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 319–338. ISBN: 978-3-030-45231-5. DOI: 10.1007/978-3-030-45231-5_17.

[NLB20] Christian A. Naesseth, Fredrik Lindsten, and David Blei. "Markovian score climbing: variational inference with KL(p||q)". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 15499–15510. ISBN: 978-1-71382-954-6. (Visited on 09/07/2023).

[VK20] Arash Vahdat and Jan Kautz. "NVAE: A deep hierarchical variational autoencoder". In: *Advances in neural information processing systems* 33 (2020), pp. 19667–19679.

[Dom21] Justin Domke. *An Easy to Interpret Diagnostic for Approximate Inference: Symmetric Divergence Over Simulations*. arXiv:2103.01030 [cs, stat]. Feb. 2021. DOI: 10.48550/arXiv.2103.01030. URL: http://arxiv.org/abs/2103.01030 (visited on 09/07/2023).

[Kin+21] Diederik Kingma et al. "Variational diffusion models". In: *Advances in neural information processing systems* 34 (2021), pp. 21696–21707.

[KTT21] Emile Krieken, Jakub Tomczak, and Annette Ten Teije. "Storchastic: A framework for general stochastic automatic differentiation". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 7574–7587.

[LBB21] Daniel Lundén, Johannes Borgström, and David Broman. "Correctness of Sequential Monte Carlo Inference for Probabilistic Programming Languages." In: *ESOP*. 2021, pp. 404–431.

[SMC21] Benjamin Sherman, Jesse Michel, and Michael Carbin. "S: computable semantics for differentiable programming with higher-order functions and datatypes". In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021), 3:1–3:31. DOI: 10.1145/3434284. URL: https://dl.acm.org/doi/10.1145/3434284 (visited on 11/10/2023).

[Sti+21] Sam Stites et al. "Learning proposals for probabilistic programs with inference combinators". en. In: *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. ISSN: 2640-3498. PMLR, Dec. 2021, pp. 1056–1066. URL: https://proceedings.mlr.press/v161/stites21a.html (visited on 09/23/2023).

[WHR21] Di Wang, Jan Hoffmann, and Thomas Reps. "Sound probabilistic inference via guide types". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 788–803.

[Zim+21] Heiko Zimmermann et al. "Nested Variational Inference". en. In: Nov. 2021. URL: https://openreview.net/forum?id=kBrHzFtwdp (visited on 11/10/2023).

[KW22] Diederik P. Kingma and Max Welling. *Auto-Encoding Variational Bayes*. arXiv:1312.6114 [cs, stat]. Dec. 2022. DOI: 10.48550/arXiv.1312.6114. URL: http://arxiv.org/abs/1312.6114 (visited on 11/02/2023).

[LCM22] Alexander K. Lew, Marco Cusumano-Towner, and Vikash K. Mansinghka. *Recursive Monte Carlo and Variational Inference with Auxiliary Variables*. arXiv:2203.02836 [cs, stat]. Nov. 2022. DOI: 10.48550/arXiv.2203.02836. URL: http://arxiv.org/abs/2203.02836 (visited on 09/07/2023).

[Mal+22] Nikolay Malkin et al. "GFlowNets and variational inference". In: *arXiv preprint arXiv:2210.00580* (2022).

[Rad+22] Alexey Radul et al. "You only linearize once: Tangents transpose to gradients". In: *arXiv preprint arXiv:2204.10923* (2022).

[Ary+23] Gaurav Arya et al. *Automatic Differentiation of Programs with Discrete Randomness*. arXiv:2210.08572 [cs, math]. Jan. 2023. DOI: 10.48550/arXiv.2210.08572. URL: http://arxiv.org/abs/2210.08572 (visited on 11/05/2023).

[LRY23] Wonyeol Lee, Xavier Rival, and Hongseok Yang. "Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), 12:335–12:366. DOI: 10.1145/3571205. URL: https://dl.acm.org/doi/10.1145/3571205 (visited on 11/10/2023).

[Lew+23a] Alexander K. Lew et al. "ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023). arXiv:2212.06386 [cs, stat], pp. 121–153. ISSN: 2475-1421. DOI: 10.1145/3571198. URL: http://arxiv.org/abs/2212.06386 (visited on 09/11/2023).

[Lew+23b] Alexander K. Lew et al. "Probabilistic Programming with Stochastic Probabilities". In: *Proceedings of the ACM on Programming Languages* 7.PLDI (June 2023), 176:1708–176:1732. DOI: 10.1145/3591290. URL: https://dl.acm.org/doi/10.1145/3591290 (visited on 09/23/2023).

[Li+23] Jianlin Li et al. "Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference". In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 1454–1482.

[LLL23] Michael Y. Li, Dieterich Lawson, and Scott Linderman. "Neural Adaptive Smoothing via Twisting". en. In: July 2023. URL: https://openreview.net/forum?id=rC6-kGN-0v (visited on 09/07/2023).