Probabilistic Programming with Vectorized Programmable Inference

MCCOY R. BECKER[*], Massachusetts Institute of Technology, USA
MATHIEU HUOT[*], Massachusetts Institute of Technology, USA
GEORGE MATHEOS, Massachusetts Institute of Technology, USA
XIAOYAN WANG, Massachusetts Institute of Technology, USA
KAREN CHUNG, Massachusetts Institute of Technology, USA
COLIN SMITH, Massachusetts Institute of Technology, USA
SAM RITCHIE, Massachusetts Institute of Technology, USA
RIF A. SAUROUS, Google, USA
ALEXANDER K. LEW, Yale University, USA
MARTIN C. RINARD, Massachusetts Institute of Technology, USA
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology, USA

We present GenJAX, a new language and compiler for vectorized programmable probabilistic inference. GenJAX integrates the vectorizing map (`vmap`) operation from array programming frameworks such as JAX into the programmable inference paradigm, enabling compositional vectorization of features such as probabilistic program traces, stochastic branching (for expressing mixture models), and programmable inference interfaces for writing custom probabilistic inference algorithms. We formalize vectorization as a source-to-source program transformation on a core calculus for probabilistic programming (λ_{GEN}), and prove that it correctly vectorizes both modeling and inference operations. We have implemented our approach in the GenJAX language and compiler, and have empirically evaluated this implementation on several benchmarks and case studies. Our results show that our implementation supports a wide and expressive set of programmable inference patterns and delivers performance comparable to hand-optimized JAX code.

CCS Concepts: • **Theory of computation → Probabilistic computation**; • **Computing methodologies →** *Machine learning algorithms*; *Massively parallel algorithms*; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: probabilistic programming, vectorization, programmable inference

ACM Reference Format:
McCoy R. Becker, Mathieu Huot, George Matheos, Xiaoyan Wang, Karen Chung, Colin Smith, Sam Ritchie, Rif A. Saurous, Alexander K. Lew, Martin C. Rinard, and Vikash K. Mansinghka. 2026. Probabilistic Programming

[*]Equal contribution.

Authors' Contact Information: McCoy R. Becker, Massachusetts Institute of Technology, Cambridge, USA, mccoyb@mit.edu; Mathieu Huot, Massachusetts Institute of Technology, Cambridge, USA, mhuot@mit.edu; George Matheos, Massachusetts Institute of Technology, Cambridge, USA, gmatheos@mit.edu; Xiaoyan Wang, Massachusetts Institute of Technology, Cambridge, USA, xyz@mit.edu; Karen Chung, Massachusetts Institute of Technology, Cambridge, USA, seoyeon@mit.edu; Colin Smith, Massachusetts Institute of Technology, Cambridge, USA, colin.smith@gmail.com; Sam Ritchie, Massachusetts Institute of Technology, Cambridge, USA, sam@perceptual.ai; Rif A. Saurous, Google, San Francisco, USA, rif@google.com; Alexander K. Lew, Yale University, New Haven, USA, alexander.lew@yale.edu; Martin C. Rinard, Massachusetts Institute of Technology, Cambridge, USA, rinard@mit.edu; Vikash K. Mansinghka, Massachusetts Institute of Technology, Cambridge, USA, vkm@mit.edu.

ACM 2475-1421/2026/1-ART87
https://doi.org/10.1145/3776729

with Vectorized Programmable Inference. *Proc. ACM Program. Lang.* 10, POPL, Article 87 (January 2026), 32 pages. https://doi.org/10.1145/3776729

1 Introduction

In recent years, probabilistic programming has demonstrated remarkable effectiveness in a range of application domains, including 3D perception and scene understanding [35, 90], probabilistic robotics [16], automated data cleaning and analysis [41, 49], particle physics [5], time series structure discovery [72, 74], test-time control of large language models [57, 58], and cognitive modeling of theory of mind [3, 4, 15, 87–89]. All of these applications require sophisticated probabilistic reasoning over complex, structured data and rely on probabilistic programming languages (PPLs) with *programmable inference* [7, 8, 19, 51, 61, 79]—the ability to customize probabilistic inference algorithms through proposals, kernels, and variational families—to improve the quality of posterior approximation. But fully realizing the benefits that probabilistic programming can deliver often requires substantial computational resources, as probabilistic inference scales by increasing the number of likelihood evaluations, sequential Monte Carlo particles, or Markov chain Monte Carlo chains.

We present GenJAX, a new language and compiler for vectorized programmable probabilistic inference. GenJAX integrates the vectorizing map (**`vmap`**) operation from array programming frameworks such as JAX [26] into the context of probabilistic programming with programmable inference, enabling the compositional vectorization of features such as probabilistic program traces, stochastic branching (for expressing mixture models), and programmable inference interfaces. This vectorization enables the implementation of compute-intensive probabilistic programming and probabilistic inference operations on modern GPUs, making it possible to deploy the substantial computational resources that GPUs provide to accelerate large-scale probabilistic inference.

Design Considerations. GenJAX is designed around the interaction between **`vmap`** and several probabilistic programming features that support the implementation of sophisticated models and inference algorithms:

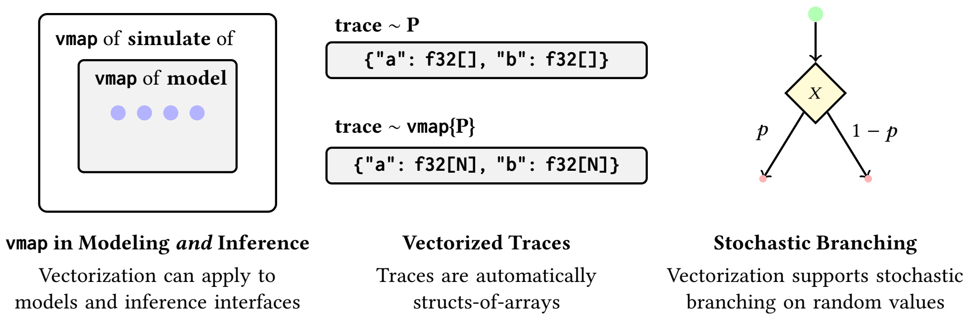
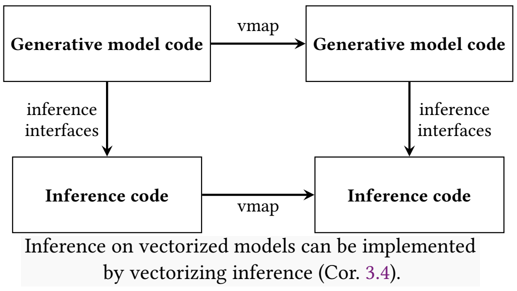

vmap of simulate of
vmap of model
trace ~ P
{"a": f32[], "b": f32[]}
trace ~ vmap{P}
{"a": f32[N], "b": f32[N]}
X
p
1 − p
vmap in Modeling *and* Inference
Vectorization can apply to models and inference interfaces
Vectorized Traces
Traces are automatically structs-of-arrays
Stochastic Branching
Vectorization supports stochastic branching on random values

Fig. 1. Computational patterns in vectorizable probabilistic programs. Left: Within models, vectorization can be used to parallelize conditionally independent computations. Within inference, vectorization can be used to simulate multiple particles in parallel. vmap should be applicable in both settings. Center: Traces are records used to represent samples from probabilistic programs. Both vectorized models and vectorized inference algorithms are designed to work with vectorized (struct-of-array) traces. Right: Probabilistic programs can branch on random values, and vmap of probabilistic programs should preserve this capability.

- **Compositional vectorization.** Our target class of probabilistic programs features multiple vectorizable computational patterns. Examples include computing likelihoods simultaneously on



GenJAX Features

Modeling & inference	Performance
Stochastic branching §2,4	Struct-of-array traces §2,4
Programmable SMC §5.2	GPU acceleration §4
Programmable MCMC §5.2	Efficient abstractions §4

Fig. 2. The design and implementation of GenJAX. Left: GenJAX extends vmap to apply to both generative models and inference algorithms. Our system implements inference on a vectorized model by vectorizing inference applied to the model, which is justified by Cor. 3.4. Right: Survey of features in our language and compiler: usage of these features illustrated in §2 and §5.2, implementation discussed in §4.

many pieces of data (as part of modeling) and evolving collections of particles (sequential Monte Carlo [20, 23, 30, 53]) or chains (Markov chain Monte Carlo [17, 29, 36, 38, 67, 81]) (as part of inference). Our integration of **vmap** must therefore support vectorization of both modeling and inference code (Fig. 1, left).

- **Vectorization of probabilistic program traces.** In many systems with programmable inference, *traces* [8, 19, 50, 60] are a key datatype: structured record objects used to represent samples. They are a data lingua franca for Monte Carlo and variational inference: traces allow the order of random variables in proposals or variational guide programs to be decoupled from the order of random variables in model programs [84]. Under vectorization by **vmap**, they support an efficient vectorized representation (*struct-of-array*, not *array-of-struct*) [70] (Fig. 1, center).
- **Vectorized stochastic branching.** Probabilistic mixture models [21], regime-switching dynamics models [25, 47, 56, 64], and adaptive inference algorithms [12, 13] all require stochastic branching using random values. GenJAX supports stochastic branching while maintaining vectorization (Fig. 1, right).

Fig. 2 presents an overview of our design and implementation.

Contributions. This paper makes the following contributions.

(1) **GenJAX: high-performance compiler (§4).** GenJAX is an open-source compiler that extends JAX and **vmap** to support programmable probabilistic inference. Probabilistic programs in GenJAX can be systematically transformed to take advantage of opportunities for vectorization in both modeling and inference. Our compiler also eliminates the overhead present in many libraries for programmable inference: we implement simulation and density interfaces using lightweight effect handlers, and exploit JAX's support for program tracing [80] to partially evaluate inference logic away at compile time, leaving only optimized array operations. Our design maintains full compatibility with JAX's underlying ecosystem for automatic differentiation (supporting algorithms like programmable variational inference [7, 9, 46, 48, 71]) and CPU/GPU/TPU compilation.

(2) **Formal model: interaction between vmap and programmable inference features (§3).** We develop a formal model characterizing how **vmap** interacts with probabilistic program traces and programmable inference interfaces. We introduce λ_{GEN}, a calculus for probabilistic programming and programmable inference, on top of a core probabilistic array language for stochastic parallel computations. We define **vmap** as a program transformation, prove its correctness, and show

how it interacts with programmable inference interfaces to support vectorization of probabilistic computations and traces.

(3) **Empirical evaluation (§5).** We evaluate our design and implementation through a series of benchmarks and case studies:
  - **Performance comparison:** We evaluate the performance characteristics of our design and implementation. GenJAX achieves near-handcoded JAX performance, and can outperform existing vectorized and high-performance PPLs and array programming frameworks (JAX [26], PyTorch [68], Pyro [8], NumPyro [69], and Gen [19]).
  - **High-dimensional vectorized inference:** We explore the performance vs. expressivity tradeoffs of our design by studying high-dimensional inference problems, including approximate *Game of Life* [28] inversion (find the previous 512 x 512 board state which leads to the observed state) and sequential 2D robot localization with simulated LIDAR measurements. In both case studies, we use GenJAX to develop sophisticated vectorized inference algorithms, including vectorized Gibbs sampling and sequential Monte Carlo with vectorized proposals. Our final GenJAX programs exhibit high approximation accuracy, and run in milliseconds on consumer-grade GPUs.

Our results demonstrate that vectorization and programmable inference abstractions can be unified through principled language and compiler design. Our system enables practitioners to write sophisticated probabilistic programs that compile to high-performance GPU code.

2 Overview

To introduce our language, consider the task of polynomial regression: given a dataset of pairs $(x_i, y_i) \in \mathbb{R}^2$, we wish to infer a polynomial relating x and y. In the following sections, we illustrate how to solve this problem using generative functions and programmable inference in GenJAX.

2.1 Vectorizing Generative Functions with `vmap`

Fig. 3 depicts a generative model for quadratic regression. The ultimate goal is to, given a noisy dataset $(x_i, y_i)_{1 \le i \le n}$, infer a quadratic function that plausibly governs the relationship between x and y. Our model for this task is defined by composing *generative functions*, each defined as a **`@gen`**-decorated Python function. The **`polynomial`** generative function describes a prior distribution on the coefficients (a, b, c) of the underlying quadratic function. Each coefficient is drawn from a standard normal distribution. A key feature of GenJAX (shared by many languages with programmable inference [8, 19, 32, 79, 86]) is that each random choice is assigned a string-valued *name*, using the syntax **`dist @ "name"`**. The **`polynomial`** generative function then returns the coefficients as a tuple. Next, the **`point`** generative function models how an individual datapoint **`y`** is generated, based on particular quadratic coefficients (a, b, c) and the corresponding input datapoint x. It computes the quadratic function's value at x, then adds a small amount of Gaussian noise. Finally, to model an entire dataset of points, **`npoint_curve`** calls **`polynomial`** to generate coefficients, and *maps* the **`point`** generative function over an input vector xs of x values, generating a vector of noisy points. This is our first use of **`vmap`** (Fig. 3, L22): we use it to generate multiple y values in parallel, exploiting the fact that the datapoints are generated *conditionally independently* of one another, given the coefficients (a, b, c). This is an instance of a general pattern that appears in many probabilistic programs, and is one key place where vectorization can yield significant speed-ups: when parts of the generative model itself can be parallelized.

Generative functions

```
1  # Basic polynomial model
2  @gen
3  def polynomial():
4    # @ denotes introduction of
5    # random choices
6    a = normal(0, 1) @ "a"
7    b = normal(0, 1) @ "b"
8    c = normal(0, 1) @ "c"
9    return (a, b, c)
10
11 # Point model with noise
12 @gen
13 def point(x, a, b, c):
14   y_mean = a + b * x + c * x ** 2
15   y = normal(y_mean, 0.2) @ "obs"
16   return y
```

Vectorization of generative functions using vmap

```
18 @gen
19 def npoint_curve(xs):
20   (a, b, c) = polynomial() @ "curve"
21   # Vectorization for modeling: here, over data points
22   ys = point.vmap(args_mapped=0)(xs, a, b, c) @ "ys"
23   return (a, b, c), ys
24
25 # Vectorized sampling from the generative function
26 # using the simulate interface.
27 xs = array([0.1, 0.3, 0.4, 0.6])
28 traces = vmap(simulate(npoint_curve), repeat=4)(xs)
29
30 # Vectorized evaluation of the pointwise density
31 # using the assess interface.
32 xs = traces.get_args()
33 densities, retvals = (
34     vmap(assess(npoint_curve), args_mapped=0)(
35         traces, xs
36     )
37 )
```

Fig. 3. Vectorization of generative functions. Left: Probabilistic programs encoding a prior over quadratic functions, and a single-datapoint likelihood. Right: vmap can be used to parallelize the likelihood: the same program that works for single points (L11-16) works for many points (L22) via vmap. Inference operations (L27, L29-36) are also compatible with vmap.

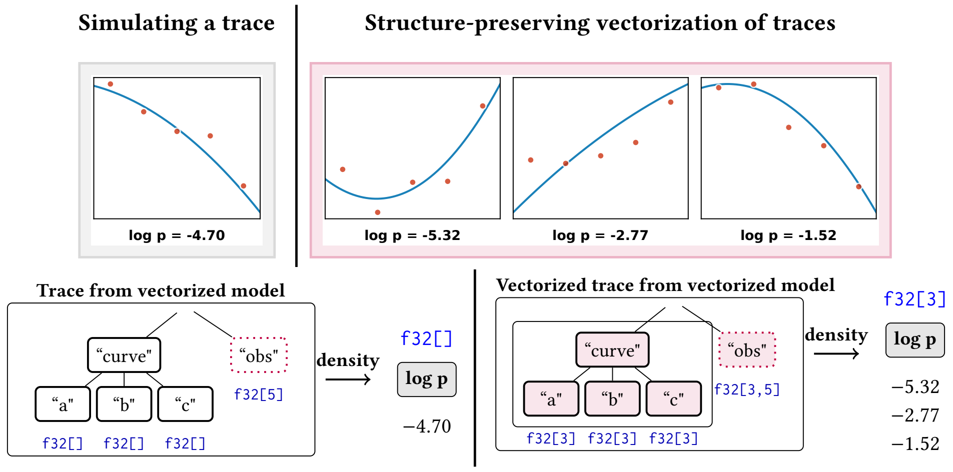


Fig. 4. Vectorized traces. Top: Traces from a single `simulate` call (left) and vectorized `vmap(simulate)` call (right) showing multiple sampled polynomial curves with varying parameters. Bottom: vmap induces a transformation on the values in the trace, shown with shape annotations. Purple outline and shading indicates random choices vectorized by vmap. The vmap operation preserves the structure of the trace, while converting scalars to arrays, returning a trace in *struct-of-array* representation.

2.2 Vectorized Programmable Inference

Generative functions are compiled to implementations of the *generative function interface* (Fig. 8), which includes methods like the following:

- **`simulate`**: runs a generative function and yields an *execution trace* (trace, for short), a record of all the named random choices encountered during execution (Fig. 4, bottom).
- **`assess`**: given a trace, computes the probability density function of the generative function's distribution at that trace (Fig. 4, **log p**).

A key idea in the design of many systems for programmable inference [7, 8, 19, 51] is that these methods can be composed to implement inference algorithms. For example, likelihood weighting involves simulating *many* possible traces from the prior, and assessing them under the likelihood. Here, we find a second key use of vectorization: by vectorizing the compiled **`simulate`** and **`assess`** methods, so that they can generate or assess many traces at once (Fig. 3, L28,34-36), we can scale the number of samples (often called *particles*) in importance sampling and sequential Monte Carlo, or the number of chains in MCMC, executing the samples or chains in parallel.

In the left pane of Fig. 5, we use the generative function interface methods to implement *one-particle* importance sampling using the **`simulate`** and **`assess`** interfaces. Importance sampling performs inference by "guessing" (sampling from a *proposal* distribution) and "checking" (scoring a guess with an *importance weight*, a ratio of the likelihood of the guess under the model to that under the proposal). The more guesses we can make, the better our posterior approximation. We can use **`vmap`** to scale the number of guesses, automatically transforming the single-particle code into a vectorized multi-particle version (Fig. 5, right pane).

With **`vmap`**, changing the *number of particles* in an inference algorithm like importance sampling changes only the array dimensions. If the algorithm is executed in parallel on a GPU, this number can be freely increased as long as the GPU has free memory. In the middle pane of Fig. 5, we illustrate the scaling behavior of vectorized importance sampling: the time remains near constant as we increase the number of particles, and the accuracy improves to convergence. This example demonstrates a common pattern when scaling vectorized inference: we can scale the vectorization to the capacity of the available GPU memory, with accuracy increasing as we use more memory. In the bottom pane of Fig. 5, we illustrate the posterior approximations constructed with different numbers of particles.

2.3 Improving Robustness Using Stochastic Branching

In real-world data, the assumptions of simple polynomial regression are often violated. Our **`polynomial`** model assumes every data point follows the same noise model—but what if 10% of our measurements follow a different distribution? The bottom left panel of Fig. 6 illustrates how inference breaks down when the model's assumptions are violated in this way. Importance sampling produces a tight fit but does not capture the explanation that we intuitively expect for the data: there is a clear quadratic trend obeyed by most of the datapoints, with a handful of outliers. The top panels of Fig. 6 show how we can improve our model's robustness by using *stochastic branching*, which allows us to account for outlier observations through heterogeneous mixture modeling. Instead of one noise model, we use stochastic branching to select between different models of the observations. The selection is based upon a random variable that we may infer from data: each data point gets a latent "outlier flag"—if true, the observation comes from a uniform distribution; if false, it follows our noisy polynomial curve. If inference works effectively, we'd expect the explanations of the data to identify the outliers and ignore them while inlier data informs the fit of our curve.

2.4 Improving Inference Accuracy Using Programmable Inference

Even when a model's assumptions are sensible, inference can fail to find good explanations of a given dataset. The middle panel of Fig. 6 shows the results of importance sampling applied to the outlier model. Importance sampling identifies likely outliers, but has wide uncertainty over the

```python
# Single particle importance sampling.
def importance_sampling(ys, xs):
  trace = simulate(default_proposal)(xs)
  logp, _ = assess(npoint_curve)(
      {"ys" : {"obs" : ys}},
      xs
  )
  w = logp - trace.get_score()
  return (trace, w)
```

```python
# Vectorized over N particles.
def vectorized_importance_sampling(ys, xs, N):
  # vmap automatically batches over n copies
  return vmap(
      importance_sampling,
      repeat=N
  )(ys, xs)

# Compute log marginal likelihood estimate.
def lmle(ws, N):
  return logsumexp(ws) - log(N)
```

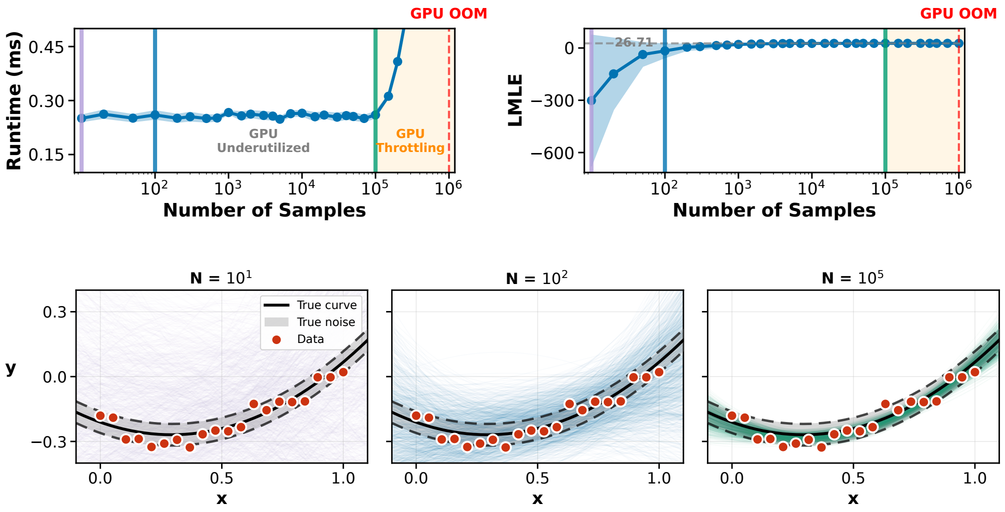


Fig. 5. Vectorized programmable inference. Top left: Single-particle importance sampling with a proposal (*default proposal* here means the prior in the `npoint_curve` model, excluding the "obs" random variable) implemented using generative function interface methods (`simulate` and `assess`). Top right: Using vmap, we can automatically transform the single-particle version into a many-particle vectorized version. Middle: The vectorized version runs in parallel on GPUs: the runtime is nearly constant as long as the GPU has memory to spare. Increasing the number of particles increases accuracy. Bottom: Posterior approximations for different numbers of particles N.

possible curves, and several curves do not seem to explain the data well. This is a kind of underfitting: by adding new latent variables to our model, we have made inference more challenging, and the "guess and check" approach of importance sampling runs into limitations, even with $N = 10^5$ particles – the limit where our GPU memory begins to saturate.

The right panel of Fig. 6 illustrates the results of a custom hybrid algorithm, which combines Gibbs sampling [29] and Hamiltonian Monte Carlo (HMC) [67]. The algorithm uses Gibbs sampling to identify which points are outliers, and HMC to sample from the posterior distribution over curves, given the inliers. As suggested by the figure, this algorithm generates much more accurate posterior samples that explain the data well. Its implementation, which we discuss next, illustrates a third opportunity for vectorization in programmable inference.

Vectorized Gibbs Sampling. In Fig. 7, we present the GenJAX implementation of the Gibbs sampling step of our hybrid algorithm. Our implementation highlights several new generative function

Robust modeling with stochastic branching

```python
# Outlier-robust observation model
@gen
def point_with_outliers(x, a, b, c):
  outlier_flag = bernoulli(0.1) @ "outlier"
  y_mean = a + b * x + c * x ** 2
  return cond(outlier_flag,
    lambda x: uniform(-2.0, 2.0),
    lambda x: trunc_norm(x, 0.05, 2.0),
    y_mean,
  ) @ "obs"
```

```python
# Vectorized curve model with outliers
@gen
def npoint_curve_with_outliers(xs):
  (a, b, c) = polynomial() @ "curve"
  ys = point_with_outliers.vmap(
        args_mapped=0,
      )(xs, a, b, c) @ "ys"
  return ys
```

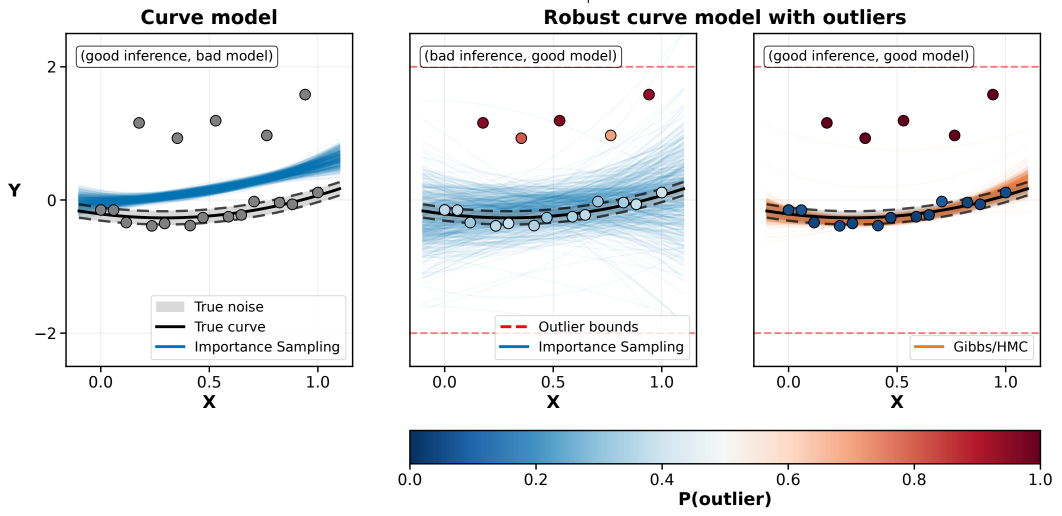


Fig. 6. Robust modeling with stochastic branching. Stochastic branching allows us to extend our models to explain more complex data, including data with outliers. Circle markers depict observed data points: the shading of the marker denotes the estimated posterior probability that the point is an outlier. Bottom, left: Using importance sampling to construct a posterior in our original model results in a poor explanation of the data. Bottom, middle: Extending the model to explicitly represent outliers as random variables should allow us to produce better explanations, but results in a harder inference problem which importance sampling can't effectively solve. Bottom, right: Changing inference to vectorized MCMC using Gibbs sampling (to infer outliers) and Hamiltonian Monte Carlo (to infer continuous parameters) finds better explanations of the data, i.e., more accurate posterior approximations.

interface methods (Fig. 8), including trace manipulation and getter methods. When we call the getter **trace.get_subtrace("ys")**, we extract the portion of the trace underneath the address **"ys"**. The call to the **update** method passes a dictionary of constraints that specifies updates to the **"outlier"** entries: **update** replays the program and splices those choices into the trace, returning a modified trace together with the incremental weight induced by the change. In our outlier model, we apply Gibbs sampling to update the vector of outlier choices **"outlier"** (which are Booleans), keeping other random choices constant. As each outlier choice is conditionally independent from the others, given all the non-outlier choices, the **"outlier"** updates can be vectorized. For each element in the **"outlier"** vector, we enumerate the unnormalized posterior density (using the generative function **assess** method) for the possible values for the outlier value, and then sample a new value from a categorical distribution, with probabilities proportional to the computed densities. Combining Gibbs sampling for discrete **"outlier"** choices with HMC for continuous

Enumerative Gibbs update for single point

```python
def gibbs_outlier(subtrace):
  def _assess(v):
    (x, a, b, c) = subtrace.args()
    chm = {"outlier": v,
           "obs": subtrace["obs"]}
    log_prob, _ = assess(point_with_outliers)(
      chm, x, a, b, c
    )
    return log_prob

  log_probs = vmap(_assess)(
    array([False, True])
  )
  return categorical(log_probs) == 1
```

Vectorized enumerative Gibbs

```python
# `trace` is a single trace object
# whose fields store batched values.
def enumerative_gibbs(trace):
  xs  = trace.get_args()
  # `subtrace` refers to the struct-of-arrays
  # view for the "ys" addresses.
  subtrace = trace.get_subtrace("ys")
  new_outliers = vmap(gibbs_outlier)(subtrace)
  # `update` applies the generative function
  # interface method that edits a trace.
  new_trace, weight, _ = update(
    trace,
    {"ys": {"outlier": new_outliers}},
  )
  return new_trace
```

Fig. 7. Vectorized enumerative Gibbs sampling for outlier detection. Left: Enumerative Gibbs update for a single data point's outlier indicator. For each possible value (inlier/outlier), we compute the log probability under the model (proportional to the unnormalized posterior) and sample a new indicator using categorical sampling. Right: Vectorized Gibbs sampling step that applies the single-point update across all data points using vmap, then updates the trace with the new outlier indicators.

simulate: *sampling*

```python
# Unconstrained sampling of a trace
tr = simulate(npoint_curve)(xs)
```

generate: *importance sampling*

```python
# Constrained sampling of a trace
partial_chm = {"ys": {"obs": data}}
tr_, weight = generate(npoint_curve)(
    partial_chm, xs
)
```

assess: *density evaluation*

```python
# Evaluate log density at traced sample
chm = get_choices(tr)
logp, retval = assess(npoint_curve)(chm, xs)
```

update: *trace modification*

```python
# Modify a trace given constraints
new_chm = {"curve": {"a": 1.0}}
tr_, w, discard = update(npoint_curve)(
    tr, new_chm, xs
)
```

Fig. 8. Generative function interface methods. GenJAX's generative functions provide several methods for programmable inference - a way to extend the system with new variants of inference using high-level interfaces. For authoring programmable algorithms which use proposal distributions (like sequential Monte Carlo), the simulate method performs unconstrained sampling and reciprocal density evaluation. For density evaluation, assess evaluates the log joint density of a generative function on traced samples. The generate interface performs constrained sampling (using importance weighting), allowing construction of a trace with observation constraints. The update method modifies a trace with provided choices, returning an updated trace and an incremental importance weight, and is used by algorithms like Gibbs sampling or Hamiltonian Monte Carlo to modify traces.

curve parameters, we designed an effective custom MCMC algorithm for inference in the outlier model, capturing an accurate posterior over curves.

3 Formal Model

In this section, we give the syntax and semantics of a core calculus for traced probabilistic programming with vectors, and formalize a program transformation that vectorizes probabilistic programs. The formal model distills key ideas from our actual implementation in JAX, described in Section 4. Figure 9 illustrates the link between implementation and core calculus.

GenJAX Implementation

```python
# Generative function definition
# with @gen decorator
@gen
def point(x, a, b, c):
    y_mean = a + b * x + c * x ** 2
    y = normal(y_mean, 0.2) @ "obs"
    return y

@gen
def npoint(xs):
    (a, b, c) = polynomial() @ "curve"
    ys = point.vmap(
        args_mapped=0,
    )(xs, a, b, c) @ "ys"
    return (a, b, c), (xs, ys)

# Core interface usage
# Sample a trace
tr = simulate(npoint, xs)

# Density evaluation
chm = get_choices(tr)
logp, retval = assess(npoint, chm, xs)
```

Syntax in λ_{GEN}

```
-- Generative function definition
-- with explicit types
point :: ℝ ⟶ ℝ^3 ⟶ G ℝ
point x (a, b, c) = do_G
  -- shorthands: + for add, × for mul
  y_m ⟵ pure (a + b × x + c × x^2)
  y ⟵ trace "obs" (normal y_m 0.2)
  return_G y

npoint :: ℝ[n] ⟶ G ℝ[n]
npoint xs = do_G
  (a, b, c) ⟵ trace "curve" polynomial
  ys ⟵ trace "ys" (
    vmap{λ x . point x (a, b, c)} xs
  )
  return_G ys

-- Core interface usage
-- Sample a trace
tr = simulate{npoint xs}

-- Density evaluation
chm, _, _ = tr
logp, retval = assess{npoint xs} chm
```

Fig. 9. GenJAX implementation vs. formal syntax. The GenJAX implementation (left) provides probabilistic programming abstractions in Python with the @gen decorator and the @ operator for addressing. The formal model (right) uses Haskell-like notation to emphasize mathematical structure: generative functions as monadic computations with type $G\,\tau$, the trace construct for recording random choices with addresses, and vmap$\{\cdot\}$, simulate$\{\cdot\}$ and assess$\{\cdot\}$ as program transformations.

3.1 Syntax of λ_{GEN}

λ_{GEN} is a simply-typed lambda calculus which extends a standard array programming calculus in two main ways:

(1) a probability monad **P** for stochastic computations; and
(2) a graded monad **G** of *generative functions*, or traced probabilistic programs.

Generative functions can be automatically compiled to the density functions and stochastic traced simulation procedures necessary for inference (Section 3.3).

Types. The types of λ_{GEN} are given at the top of Figure 10, and comprise:

- *Ground types.* We define representative base types: booleans $\mathbb{B}$, real numbers $\mathbb{R}$ and positive real numbers $\mathbb{R}_{>0}$. A batched type T is a base type B or a tensor type $T[n]$. A tensor type $T[n]$ is an n-fold product of a type T, representing an n-dimensional array of elements of type T. The ground types η consist of batched types T, product types $\eta_1 \times \eta_2$, the unit type 1, and string-indexed record types $\{k_1 : \eta_1, \ldots, k_n : \eta_n\}$, where all keys k_i are distinct. The type of empty record is also noted 1.
- *Density-carrying distributions.* The type $\mathbf{D}\,\eta$ represents density-carrying distributions over the ground type η. We assume that for each primitive distribution $d : \eta_1 \longrightarrow \mathbf{D}\,\eta_2$, we have an

$$
\begin{array}{rl}
\text{Base types } B & ::= \mathbb{B} \mid \mathbb{R} \mid \mathbb{R}_{>0} \\
\text{Batched types } T & ::= B \mid T[n]
\end{array}
\qquad
\begin{array}{rl}
\text{Ground types } \eta & ::= 1 \mid T \mid \eta_1 \times \eta_2 \mid \{k_1 : \eta_1, \dots, k_n : \eta_n\} \\
\text{Types } \tau & ::= \eta \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \mathbf{D}\ \eta \mid \mathbf{P}\ \eta \mid \mathbf{G}_\gamma\ \eta
\end{array}
$$

$$
\begin{array}{rl}
\text{Grading } \gamma & ::= \{k_1 : \eta_1, \dots, k_n : \eta_n\} \\
\text{Terms } t & ::= () \mid c \mid p \mid x \mid (t_1, t_2) \mid \pi_i t \\
& \mid t[k] \mid \{k_1 : t_1, \dots, k_n : t_n\} \\
& \mid t_1\ t_2 \mid \lambda x.t \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
& \mid \mathbf{select}(t_1, t_2, t_3) \mid \mathbf{trace}(s, t) \\
& \mid \mathbf{return}_G\ t \mid \mathbf{return}_P\ t \\
& \mid \mathbf{do_G}\{m\} \mid \mathbf{do_P}\{m\} \mid \mathbf{sample}\ t
\end{array}
\qquad
\begin{array}{rl}
\text{Monadic } m & ::= t \mid x \leftarrow t; m \\
\text{Constants } c & ::= a\ (\in T) \\
\text{Primitives } p & ::= \text{Scalar} \mid \text{Vectorized} \mid \text{Array} \mid \text{Distribution} \\
\text{Scalar} & ::= \cos \mid \sin \mid \exp \mid \text{add} \mid \text{mul} \\
\text{Vectorized} & ::= \text{dot} \mid \text{svd} \mid \text{sum} \\
\text{Array} & ::= \text{fold} \mid \text{scan} \mid \text{reduce} \\
\text{Distribution} & ::= \mathbf{uniform} \mid \mathbf{normal} \mid \mathbf{bernoulli}
\end{array}
$$

$$
\frac{\Gamma \vdash t : \mathbf{P}\ \eta \quad \Gamma, x : \eta \vdash \mathbf{do_P}\{m\} : \mathbf{P}\ \eta'}{\Gamma \vdash \mathbf{do_P}\{x \leftarrow t; m\} : \mathbf{P}\ \eta'}
\qquad
\frac{\Gamma \vdash t : \mathbf{D}\ \eta}{\Gamma \vdash \mathbf{sample}\ t : \mathbf{P}\ \eta}
\qquad
\frac{\Gamma \vdash t : \{k_1 : \eta_1, \dots, k_n : \eta_n\} \quad k = k_i}{\Gamma \vdash t[k] : \eta_i}
$$

$$
\frac{\Gamma \vdash t_1 : \eta_1 \quad \dots \quad \Gamma \vdash t_n : \eta_n}{\Gamma \vdash \{k_1 : t_1, \dots, k_n : t_n\} : \{k_1 : \eta_1, \dots, k_n : \eta_n\}}
\qquad
\frac{\Gamma \vdash t : \eta}{\Gamma \vdash \mathbf{return}_P\ t : \mathbf{P}\ \eta}
\qquad
\frac{\Gamma \vdash t_1 : \mathbb{B}^s \quad \Gamma \vdash t_2 : T^s \quad \Gamma \vdash t_3 : T^s}{\Gamma \vdash \mathbf{select}(t_1, t_2, t_3) : T^s}
$$

$$
\frac{\Gamma \vdash t_1 : T_1[n] \quad \Gamma \vdash t_2 : T_1 \to T_2 \to T_2 \quad \Gamma \vdash t_3 : T_2}{\Gamma \vdash \text{fold}(t_1, t_2, t_3) : T_2}
\qquad
\frac{\Gamma \vdash t : \mathbf{P}\ \eta}{\Gamma \vdash \mathbf{do_P}\{t\} : \mathbf{P}\ \eta}
\qquad
\frac{\Gamma \vdash t : \mathbf{G}_\gamma\ \eta}{\Gamma \vdash \mathbf{do_G}\{t\} : \mathbf{G}_\gamma\ \eta}
$$

$$
\frac{\Gamma \vdash t_1 : T_1[n] \quad \Gamma \vdash t_2 : T_1 \to T_2 \to T_2 \quad \Gamma \vdash t_3 : T_2}{\Gamma \vdash \text{scan}(t_1, t_2, t_3) : (T_2[n] \times T_2)}
\qquad
\frac{k \in \text{Str} \quad \Gamma \vdash t : \mathbf{G}_\gamma\ \eta}{\Gamma \vdash \mathbf{trace}(k, t) : \mathbf{G}_{\{k \mapsto \gamma\}}\ \eta}
\qquad
\frac{k \in \text{Str} \quad \Gamma \vdash t : \mathbf{D}\ \eta}{\Gamma \vdash \mathbf{trace}(k, t) : \mathbf{G}_{\{k \mapsto \eta\}}\ \eta}
$$

$$
\frac{\Gamma \vdash t : \eta}{\Gamma \vdash \mathbf{return}_G\ t : \mathbf{G}_{\{\}}\ \eta}
\qquad
\frac{\Gamma \vdash t : \mathbf{G}_\gamma\ \eta \quad \Gamma, x : \eta \vdash \mathbf{do_G}\{m\} : \mathbf{G}_{\gamma'}\ \eta' \quad \mathit{keys}(\gamma) \cap \mathit{keys}(\gamma') = \emptyset}{\Gamma \vdash \mathbf{do_G}\{x \leftarrow t; m\} : \mathbf{G}_{\gamma \mathbin{+\!\!+} \gamma'}\ \eta'}
$$

Fig. 10. Syntax and typing rules of λ_{GEN}

additional density primitive $d.density : \eta_1 \to \eta_2 \to \mathbb{R}$ that computes the associated probability density (or mass) function with respect to the *stock measure* on η_2.[1]

- *Stochastic computations.* The type $\mathbf{P}\ \eta$ is used to track computations that use probabilistic sampling. We use a Haskell-like do notation. $\mathbf{do_P}\{x \leftarrow t; m\}$ sequences probabilistic computations, where x is bound to the result of t in the continuation m. $\mathbf{return}_P\ t$ embeds a deterministic computation t as a probabilistic one, enabling deterministic logic within probabilistic computation. $\mathbf{sample}\ t$ samples from a given primitive distribution.
- *Traced generative functions.* The types $\mathbf{G}_\gamma\ \eta$ represent traced generative functions. Here, γ is a *grading* tracking the type of the generative function's trace [50]. The grading is a record type $\{k_1 : \eta_1, \dots, k_n : \eta_n\}$, where the keys k_i track the names supplied to $\mathbf{trace}(\cdot, \cdot)$ calls in the probabilistic program, and the corresponding η_i is the type of data traced by each call. Note that when a generative function traces a call to another generative function, the corresponding η_i will itself be a string-keyed record, leading to the sorts of hierarchical traces depicted in Fig. 4. We equip the grading with a monoid structure $+\!\!+$ for concatenation. It merges two records in an obvious way. The unit of the monoid is the empty record $\{\}$ of type 1. This turns $\mathbf{G}_\gamma\ \eta$ into a graded monad, where sequencing two generative function programs via $\mathbf{do_G}\{\}$ concatenates their trace types. Note that we restrict the return type η of a generative function to be ground.
- *Functions.* We also have standard function types $\tau_1 \to \tau_2$.

[1]These stock measures are defined in the standard way, by induction on η_2. At continuous base types (e.g. $\mathbb{R}$), we choose the Lebesgue measure, and at discrete base types (e.g. $\mathbb{B}$), the counting measure. Products of base types, including product types, record types, and tensor types, have as stock measures the products of the stock measures of their constituent types.

Tensor Shape Notation. A tuple $(s_1, \ldots, s_k)$ of natural numbers is called a *tensor shape*. For any batched type T, we write T^s as shorthand for $T[s_k] \ldots [s_1]$.

A shape t has *prefix shape* s if $s := (s_1, \ldots, s_k)$ and $t := (s_1, \ldots, s_k, t_1, \ldots, t_m)$. *Shape concatenation*: if $s_1 := (s_1^1, \ldots, s_k^1)$ and $s_2 := (s_1^2, \ldots, s_j^2)$, then $s_1 + s_2 := (s_1^1, \ldots, s_k^1, s_1^2, \ldots, s_j^2)$. *Shape subtraction*: given shapes s, t where t has prefix shape s, we write $t - s$ for the unique shape s_2 such that $s + s_2 = t$. For shape $s = (s_1, \ldots, s_k)$, we write $i \in s$ to mean $i \in \{(i_1, \ldots, i_k) \mid \forall j, 1 \leq i_j \leq s_j\}$. We also use this syntactic notation for sets and functions. Given set X and shape $s = (s_1, \ldots, s_k)$, we write X^s for $X^{s_1 \times \ldots \times s_k}$. Note that $(X^q)^s = X^{s+q}$. For $x \in X^s$ and $i \in s$, we write $x[i]$ for the i-th projection of x. For $x \in X^t$ where t has prefix shape s and $i \in s$, we extend $x[i]$ to denote an element of X^{t-s}. For function $f : X \to Y$ and shape s, we write $f_s : X^s \to Y^s$ for pointwise application: $(f_s(x))[i] = f(x[i])$. This extends to multi-argument functions: $f_s : X_1^s \times \ldots \times X_k^s \to Y_1^s \times \ldots \times Y_m^s$.

Terms. The terms of λ_{GEN} and their typing rules are given in the middle and bottom parts of Figure 10. Terms include:

- *String-indexed record literals* $\{k_1 : t_1, \ldots, k_n : t_n\}$, which create a record with keys $k_1, \ldots, k_n$ and associated values computed by the terms $t_1, \ldots, t_n$. If t is of record type, $t[k]$ retrieves the value associated with key k.
- *Constants.* Constants c include base values $a \in B$ for every base type B and for every tensor type T. A value of type $B[s_k]...[s_1]$ is a k-dimensional array of type B. The tuple $s := (s_1, \ldots, s_k)$ is called the tensor shape of the tensor.
- *Scalar Primitives.* Scalar primitives include elementwise operations such as cos, exp, and mul.
- *Vectorized Primitives.* Vectorized primitives include operations that operate across tensor dimensions, such as the dot product ($\text{dot}_T : T \to T \to \mathbb{R}$, where T has base type $\mathbb{R}$), singular value decomposition ($\text{svd}_{n,m} : \mathbb{R}[m][n] \to \mathbb{R}[m][n] \times \mathbb{R}[m][n] \times \mathbb{R}[m][n]$), and summation ($\text{sum}_T : T \to \mathbb{R}$, where T has base type $\mathbb{R}$).
- *Array Primitives.* Array primitives include operations that operate on arrays, such as fold, scan, and reduce. fold repeats a binary function over an array, scan further returns the intermediate results, and reduce is a parallel version of fold that assumes the operation to be associative.
- *Batched primitives.* We assume that every primitive (scalar, vectorized, array, and distribution) p can be subscripted with a tensor shape s to obtain p_s, representing a batched version of the primitive that is applied element-wise. For instance, for the scalar primitive cos we have $\cos_s : \mathbb{R}^s \to \mathbb{R}^s$. Given a vectorized primitive such as $\text{dot}_{\mathbb{R}[n]}$, $\text{dot}_{\mathbb{R}[n],s} : \mathbb{R}[n]^s \times \mathbb{R}[n]^s \to \mathbb{R}^s$ (for tensor shapes s) represents a batched version of the primitive $\text{dot}_{\mathbb{R}[n]}$.
- *Distribution primitives.* The language provides built-in distribution constructors: $\textbf{uniform} : \mathbf{D}\,\mathbb{R}$ is the uniform distribution over $(0, 1)$, $\textbf{normal} : \mathbb{R} \times \mathbb{R}_{>0} \to \mathbf{D}\,\mathbb{R}$ is the normal distribution with mean and variance parameters, and $\textbf{bernoulli} : \mathbb{R} \to \mathbf{D}\,\mathbb{B}$ is the Bernoulli distribution with a probability parameter. Batched versions of distribution primitives generate tensors of independent samples. For instance, for a distribution constructor such as **uniform**, we have $\textbf{uniform}_s : \mathbf{D}\,\mathbb{R}^s$ returning independent samples from the uniform distribution on $(0, 1)$ in a tensor of shape s.
- *Traced programs.* Traced programs are written in a monadic style similar to $\textbf{do}_\mathbf{P}\{\}$. The key change is that all sampled variables must be named, and are accumulated into a trace. We write Str for the set of strings. Primitive distributions $t : \mathbf{D}\,\eta$ can be sampled using the syntax $\textbf{trace}(k, t) : \mathbf{G}_{\{k \mapsto \eta\}}\,\eta$; the resulting program returns the sampled value and records it in the trace with name $k \in$ Str. Compound generative functions $t : \mathbf{G}_\gamma\,\eta$ can also be arguments to **trace**: in this case $\textbf{trace}(k, t)$ has type $\mathbf{G}_{\{k \mapsto \gamma\}}\,\eta$. Note that in the trace type, the entire trace type γ of t has been nested under the name k. Deterministic computations can be embedded into **G** with the syntax $\textbf{return}_G\, t : \mathbf{G}_{\{\}}\,\eta$; the resulting programs have empty traces. $\textbf{do}_\mathbf{G}\{x \leftarrow t; m\}$

Types

$$
\begin{array}{llllll}
\llbracket \mathbb{B} \rrbracket & = \mathbb{B} & \llbracket \eta_1 \times \eta_2 \rrbracket & = \llbracket \eta_1 \rrbracket \times \llbracket \eta_2 \rrbracket & \llbracket \mathbf{P}\,\eta \rrbracket & = \mathcal{P}\llbracket \eta \rrbracket \\
\llbracket \mathbb{R} \rrbracket & = \mathbb{R} & \llbracket \{k_1 : \eta_1, \ldots, k_n : \eta_n\} \rrbracket & = \llbracket \eta_1 \rrbracket \times \cdots \times \llbracket \eta_n \rrbracket & \llbracket \mathbf{D}\,\eta \rrbracket & = \mathcal{P}_{\ll}\llbracket \eta \rrbracket \\
\llbracket T[n] \rrbracket & = \llbracket T \rrbracket^n & \llbracket \tau_1 \to \tau_2 \rrbracket & = [\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket] & \llbracket \mathbf{G}_\gamma\,\eta \rrbracket & = \mathcal{P}_{\ll}\llbracket \gamma \rrbracket \times [\llbracket \gamma \rrbracket, \llbracket \eta \rrbracket] \\
\llbracket 1 \rrbracket & = 1 & \llbracket \tau_1 \times \tau_2 \rrbracket & = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket & &
\end{array}
$$

Terms

$$
\begin{array}{llll}
\llbracket () \rrbracket(\gamma) & = () & \llbracket t_1\ t_2 \rrbracket(\gamma) & = \llbracket t_1 \rrbracket(\gamma)\ \llbracket t_2 \rrbracket(\gamma) \\
\llbracket p_s \rrbracket(\gamma) & = p_s & \llbracket (t_1, t_2) \rrbracket(\gamma) & = (\llbracket t_1 \rrbracket(\gamma), \llbracket t_2 \rrbracket(\gamma)) \\
\llbracket x : \tau \rrbracket(\gamma) & = \gamma(x) & \llbracket \mathbf{return}_G\ t \rrbracket(\gamma) & = (\delta_{\{\}}, \lambda().\llbracket t \rrbracket(\gamma)) \\
\llbracket a \rrbracket(\gamma) & = a & \llbracket \mathbf{trace}(k, t : \mathbf{G}_{\gamma'}\ \tau) \rrbracket(\gamma) & = \llbracket t \rrbracket(\gamma) \\
\llbracket t[k] \rrbracket(\gamma) & = (\llbracket t \rrbracket(\gamma))_k & \llbracket \mathbf{trace}(k, t : \mathbf{D}\ \eta) \rrbracket(\gamma) & = (\llbracket t \rrbracket(\gamma), \lambda x.x) \\
\llbracket \pi_i t \rrbracket(\gamma) & = (\llbracket t \rrbracket(\gamma))_i & \llbracket \lambda x : \tau.t \rrbracket(\gamma) & = \lambda x : \llbracket \tau \rrbracket.\llbracket t \rrbracket(\gamma) \\
\llbracket \mathbf{do_P}\{t\} \rrbracket(\gamma) & = \llbracket t \rrbracket(\gamma) & \llbracket \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \rrbracket(\gamma) & = \llbracket t_2 \rrbracket(\gamma[x \mapsto \llbracket t_1 \rrbracket(\gamma)]) \\
\llbracket \mathbf{return}_P\ t \rrbracket(\gamma) & = \delta_{\llbracket t \rrbracket(\gamma)} & \llbracket \mathbf{select}(t_1, t_2, t_3) \rrbracket(\gamma) & = \mathbf{select}(\llbracket t_1 \rrbracket(\gamma), \llbracket t_2 \rrbracket(\gamma), \llbracket t_3 \rrbracket(\gamma)) \\
\llbracket \mathbf{sample}\ t \rrbracket(\gamma) & = \llbracket t \rrbracket(\gamma) & \llbracket \{k_1 : t_1, \ldots, k_m : t_m\} \rrbracket(\gamma) & = (\llbracket t_1 \rrbracket(\gamma), \ldots, \llbracket t_m \rrbracket(\gamma)) \\
\llbracket \mathbf{do_G}\{t\} \rrbracket(\gamma) & = \llbracket t \rrbracket(\gamma) & \llbracket \mathbf{do_P}\{x \leftarrow t; m\} \rrbracket(\gamma, A) & = \int \llbracket t \rrbracket(\gamma, du) \llbracket m \rrbracket(\gamma[x \mapsto u], A)
\end{array}
$$

$$
\begin{array}{lll}
\llbracket \mathbf{do_G}\{x \leftarrow t; m\} \rrbracket_1(\gamma, A) & = & \int \llbracket t \rrbracket_1(\gamma, du) \int \llbracket \mathbf{do_G}\{m\} \rrbracket(\gamma[x \mapsto \llbracket t \rrbracket_2(\gamma)(u)], dv) \delta_{u +\!\!+ v}(A) \\
\llbracket \mathbf{do_G}\{x \leftarrow t; m\} \rrbracket_2(\gamma) & = & \lambda tr.\llbracket \mathbf{do_G}\{m\} \rrbracket_2(\gamma[x \mapsto \llbracket t \rrbracket_2(\gamma)(\pi_{grade(t)}(tr))])(\pi_{grade(\mathbf{do_G}\{m\})}(tr))
\end{array}
$$

Fig. 11. Denotational semantics of λ_{GEN}

can be used for sequencing, but the top-level names used in t and m must be disjoint. Using the $\mathbf{trace}(k, t)$ construct to nest a call to a generative function under a new label k is one way to ensure disjointness even when the same subprogram is invoked multiple times.

- *Other terms.* We also have the standard terms of the λ-calculus, e.g. the unit value $()$, variables x, abstractions $\lambda x.t$, applications $t_1\ t_2$, tuples (t_1, t_2), projections $\pi_i t$. We write $\mathbf{select}(t_1, t_2, t_3)$ for the conditional selection of elements from a tensor. The three subterms must have batched types of the same shape, and the returned value at index i is $t_2[i]$ if $t_1[i]$ is true, $t_3[i]$ otherwise.

3.2 Denotational Semantics

Figure 11 gives a denotational semantics for λ_{GEN} using quasi-Borel spaces (QBS) [39], a standard mathematical framework for higher-order probabilistic programming. See the supplementary material for the definition of QBS. We assign to each type τ a space $\llbracket \tau \rrbracket$ and to each term $\Gamma \vdash t : \tau$ a map $\llbracket t \rrbracket : \prod_{x \in \Gamma} \llbracket \Gamma(x) \rrbracket \to \llbracket \tau \rrbracket$ from the interpretation of the environment to the interpretation of its return type. We use $[X, Y]$ to denote the quasi-Borel function space, $X \times Y$ for the product, 1 for a singleton QBS, and denote by $\mathcal{P}$ the probability monad on QBS (see, e.g., Heunen et al. [39]). We also use $\otimes$ for the product measure, and δ_x for the Dirac measure at x. Base types are interpreted as their usual sets equipped with the Borel-sigma algebra (random elements are measurable functions). All our ground types are interpreted as standard Borel spaces, and we denote by $\mathcal{P}_{\ll}\llbracket \eta \rrbracket$ the space of probability measures on the standard Borel space $\llbracket \eta \rrbracket$ that are absolutely continuous w.r.t. the stock measure for type η. A generative function of type $\mathbf{G}_\gamma\ \eta$ is interpreted as a pair of a measure on γ that is absolutely continuous w.r.t. the stock measure on $\llbracket \gamma \rrbracket$, and a return value function $\llbracket \gamma \rrbracket \to \llbracket \eta \rrbracket$ which computes the program's return value given a trace, i.e. given values for all the random choices in the program. If $\llbracket t \rrbracket$ denotes a tuple, such as when $t : \mathbf{G}_\gamma\ \eta$, we write $\llbracket t \rrbracket_k$ for its k-th component. For a trace $tr \in \llbracket \gamma +\!\!+ \gamma' \rrbracket$, we write $\pi_\gamma(tr)$ and $\pi_{\gamma'}(tr)$ the projections to $\llbracket \gamma \rrbracket$, $\llbracket \gamma' \rrbracket$ respectively. For a term t of type $\mathbf{G}_\gamma\ \eta$, we write $grade(t)$ to extract the grade γ.

Programmable inference transformations on types

$$\begin{array}{llll}
\mathbf{simulate}\{\mathbf{D}\ \eta\} & = \mathbf{P}\ (\eta \times \mathbb{R}) & \mathbf{assess}\{\mathbf{D}\ \eta\} & = \eta \to \mathbb{R} \\
\mathbf{simulate}\{\mathbf{G}_\gamma\ \eta\} & = \mathbf{P}\ (\gamma \times \eta \times \mathbb{R}) & \mathbf{assess}\{\mathbf{G}_\gamma\ \eta\} & = \gamma \to \eta \times \mathbb{R}
\end{array}$$

Transformations act homomorphically on product and function types and leave ground types unchanged (e.g. $\mathbf{assess}\{\tau_1 \to \tau_2\} = \mathbf{assess}\{\tau_1\} \to \mathbf{assess}\{\tau_2\}$).

On terms

$$\begin{array}{ll}
\mathbf{simulate}\{\mathbf{bernoulli}\} & = \lambda p.\mathbf{do_P}\{b \leftarrow \mathbf{sample}\ (\mathbf{bernoulli}\ p); \mathbf{return}_P\ (b, \mathbf{select}(b, p, 1-p))\} \\
\mathbf{simulate}\{\mathbf{return}_G\ t\} & = \mathbf{return}_P\ (\{\}, \mathbf{simulate}\{t\}, 1) \\
\mathbf{simulate}\{\mathbf{trace}(k, t : \mathbf{D}\ \eta)\} & = \mathbf{do_P}\{(x, r) \leftarrow \mathbf{simulate}\{t\}; \mathbf{return}_P\ (\{k : x\}, x, r)\} \\
\mathbf{simulate}\{\mathbf{trace}(k, t : \mathbf{G}_\gamma\ \eta)\} & = \mathbf{do_P}\{(u, x, r) \leftarrow \mathbf{simulate}\{t\}; \mathbf{return}_P\ (\{k : u\}, x, r)\} \\
\mathbf{simulate}\{\mathbf{do_G}\{x \leftarrow t; m\}\} & = \mathbf{do_P}\{(u, x, w) \leftarrow \mathbf{simulate}\{t\}; (u', y, w') \leftarrow \mathbf{simulate}\{\mathbf{do_G}\{m\}\}; \\
 & \qquad \mathbf{return}_P\ (u \mathbin{+\!\!+} u', y, w \cdot w')\} \\
\mathbf{assess}\{\mathbf{bernoulli}\} & = \lambda p.\lambda b.\mathbf{select}(b, p, 1-p) \\
\mathbf{assess}\{\mathbf{return}_G\ t\} & = \lambda u.(\mathbf{assess}\{t\}, 1) \\
\mathbf{assess}\{\mathbf{trace}(k, t : \mathbf{D}\ \eta)\} & = \lambda u.(u[k], \mathbf{assess}\{t\}(u[k])) \\
\mathbf{assess}\{\mathbf{trace}(k, t : \mathbf{G}_\gamma\ \eta)\} & = \lambda u.\mathbf{assess}\{t\}(u[k]) \\
\mathbf{assess}\{\mathbf{do_G}\{x \leftarrow t; m\}\} & = \lambda u.\mathbf{let}\ (x, w) = \mathbf{assess}\{t\}(\pi_{grade(t)}(u))\ \mathbf{in} \\
 & \qquad\qquad \mathbf{let}\ (y, w') = \mathbf{assess}\{\mathbf{do_G}\{m\}\}(\pi_{grade(\mathbf{do_G}\{m\})}(u))\ \mathbf{in}(y, w \cdot w')
\end{array}$$

Transformations act analogously to **bernoulli** on other primitive distributions, and homomorphically on terms introducing or eliminating products and functions (e.g., $\mathbf{assess}\{(t_1, t_2)\} = (\mathbf{assess}\{t_1\}, \mathbf{assess}\{t_2\})$).

Fig. 12. Definitions of the $\mathbf{simulate}\{\cdot\}$ and $\mathbf{assess}\{\cdot\}$ transformations, on types and terms.

3.3 Programmable Inference

Generative functions support methods $\mathbf{simulate}\{-\}$ and $\mathbf{assess}\{-\}$, which are implemented as source-to-source program transformations. We present the transformations, which are standard [50, 54], in Figure 12. At a high level, $\mathbf{simulate}\{-\}$ allows us to run the program and simulate traces. It returns a trace of the program, a return value, and the joint density at that sampled trace. $\mathbf{assess}\{-\}$ returns the density of a given trace. It does so by running the program, but with each primitive sampling statement replaced by code that looks up the pre-determined outcome in the given trace, and multiplies the density of the primitive distribution into a running total. The key correctness property of $\mathbf{simulate}\{-\}$ and $\mathbf{assess}\{-\}$ is given in Proposition 3.1, which can be proved using a standard logical relations argument analogous to the ones given in, e.g., [7, 51, 54]. We denote by $f_*\mu$ the pushforward distribution of μ by f.

Proposition 3.1. *Let $\vdash t : \mathbf{G}_\gamma\ \eta$ be a closed term of generative function type, with denotation $(\mu, f) = [\![t]\!]$. Further, let ν be the stock measure associated with the record type γ. Then:*

- $[\![\mathbf{simulate}\{t\}]\!] = \langle id, f, \frac{d\mu}{d\nu}\rangle_*\mu$ *(i.e., $\mathbf{simulate}\{t\}$ faithfully generates a trace u from μ, and returns $(u, f(u), w)$, where w is the density of μ at u); and*
- $[\![\mathbf{assess}\{t\}]\!] = \langle f, \frac{d\mu}{d\nu}\rangle$ *(i.e., $\mathbf{assess}\{t\}$ faithfully computes the return value function and the density of μ at a given trace).*

3.4 Vectorization Program Transform

We introduce $\mathbf{vmap}_n\{-\}$ as a program transform for vectorization. $\mathbf{vmap}_n\{-\}$ takes an integer n and a term t of type τ and returns a vectorized version of type $\tau[n]$, defined inductively as follows:

Types and contexts

$$\mathbf{vmap}_n\{\tau\} \quad = \tau[n] \qquad \mathbf{vmap}_n\{x_1 : \tau_1, \dots, x_k : \tau_k\} \quad = x_1 : \tau_1[n], \dots, x_k : \tau_k[n]$$

Terms

$$\mathbf{vmap}_n\{()\} \quad = () \qquad \mathbf{vmap}_n\{a : T\} \quad = \bar{a}^n : T[n] \qquad \mathbf{vmap}_n\{p_s\} \quad = p_{(n)+s}$$

$\mathbf{vmap}_n\{\cdot\}$ acts homomorphically on other terms (e.g., $\mathbf{vmap}_n\{t_1\ t_2\} = \mathbf{vmap}_n\{t_1\}\ \mathbf{vmap}_n\{t_2\}$

Fig. 13. Inductive definition of $\mathbf{vmap}_n\{\cdot\}$, a vectorization transformation, on types and terms.

$$\begin{array}{ll}
R_1 & := \{((x, \dots, x), x) \mid x \in [\![1]\!]\} \\
R_B & := \{((x_1, \dots, x_n), x) \mid x_i \in [\![B]\!],\ x = (x_1, \dots, x_n)\} \\
R_T & := \{((x_1, \dots, x_n), x) \mid x_i \in [\![T]\!],\ x = (x_1, \dots, x_n)\} \\
R_{\tau_1 \times \tau_2} & := \{(((x_1, y_1), \dots, (x_n, y_n)), (z_1, z_2)) \mid ((x_1, \dots, x_n), z_1) \in R_{\tau_1}, ((y_1, \dots, y_n), z_2) \in R_{\tau_2}\} \\
R_{\tau_1 \to \tau_2} & := \{((f_1, \dots, f_n), g) \mid \forall ((x_1, \dots, x_n), y) \in R_{\tau_1},\ ((f_1(x_1), \dots, f_n(x_n)), g) \in R_{\tau_2}\} \\
R_{\{k_1:\eta_1, \dots, k_m:\eta_m\}} & := \{((x_1, \dots, x_n), y) \mid \forall j \in \{1, \dots, m\},\ ((\pi_j x_1, ..., \pi_j x_n), \pi_j y) \in R_{\eta_j}\} \\
R_{\mathbf{D}\ \eta} & := \{((\mu_1, \dots, \mu_n), \mu) \mid \mu = R_{\eta_*}(\bigotimes_{i=1}^n \mu_i)\} \\
R_{\mathbf{P}\ \eta} & := \{((\mu_1, \dots, \mu_n), \mu) \mid \mu = R_{\eta_*}(\bigotimes_{i=1}^n \mu_i)\} \\
R_{\mathbf{G}_\gamma\ \eta} & := \{(((\mu_1, f_1), \dots, (\mu_n, f_n)), (\nu, g)) \mid ((\mu_1, \dots, \mu_n), \nu) \in R_{\mathbf{P}\ \gamma}, ((f_1, \dots, f_n), g) \in R_{\gamma \to \eta}\}
\end{array}$$

Fig. 14. Logical relations for establishing the correctness of vmap.

$$\begin{array}{rlrl}
(\mathbf{D}\ \eta)[n] & ::= \mathbf{D}\ \eta[n] & (\tau_1 \to \tau_2)[n] & ::= \tau_1[n] \to \tau_2[n] \\
(\mathbf{G}_\gamma\ \eta)[n] & ::= \mathbf{G}_{\gamma[n]}\ \eta[n] & (\tau_1 \times \tau_2)[n] & ::= \tau_1[n] \times \tau_2[n] \\
(\mathbf{P}\ \eta)[n] & ::= \mathbf{P}\ \eta[n] & 1[n] & ::= 1 \\
\{k_1 : \eta_1, \dots, k_n : \eta_n\}[n] & ::= \{k_1 : \eta_1[n], \dots, k_n : \eta_n[n]\} & (T)[n] & ::= T[n]
\end{array}$$

On primitives, $\mathbf{vmap}_n\{-\}$ will simply extend the batching shape s of the primitive to $(n) + s$. For instance, $\mathbf{vmap}_n\{\text{add}_s\} : \mathbb{R}^{(n)+s} \to \mathbb{R}^{(n)+s} \to \mathbb{R}^{(n)+s}$ will be the elementwise addition of two tensors of shape $(n) + s$. $\mathbf{vmap}_n\{-\}$ performs an automatic "array of struct" to "struct of array" conversion, and extends homomorphically to all the constructs of the language. We present $\mathbf{vmap}_n\{-\}$ as a program transformation in Figure 13. If a is a tensor literal of shape s, we denote by $\bar{a}^n$ the tensor literal of shape $(n) + s$ that consists of n copies of a.

To prove the correctness of $\mathbf{vmap}_n\{\}$, we use a proof by logical relations. In Fig. 14, we define relations $R_\tau \subseteq [\![\tau]\!]^n \times [\![\tau[n]]\!]$ for all types τ, which intuitively encode the requirement for a value of type $\tau[n]$ to be a correct vectorization of n distinct values of type τ. We denote by $R_{\eta_*}\mu$ the pushforward of the distribution μ by the *functional* relation R_η.[2] This relies on the fact, which can be established via a simple inductive argument, that for ground types η our logical relations are functional. Having defined these logical relations, we establish the fundamental lemma by induction on the structure of the program.

Proposition 3.2 (Fundamental lemma for $\mathbf{vmap}_n\{\cdot\}$). *Let $n \in \mathbb{N}$ and $x_1 : \tau_1, \dots, x_m : \tau_m \vdash t : \tau$. If $((v_j^1, \dots, v_j^n), w_j) \in R_{\tau_j}$ for each $1 \leq j \leq m$, and if $\gamma^i := (x_1 \mapsto v_1^i, \dots, x_m \mapsto v_m^i)$ for each $1 \leq i \leq n$, and $\gamma' := (x_1 \mapsto w_1, \dots, x_m \mapsto w_m)$, then*

$$(([\![t]\!](\gamma^1), \dots, [\![t]\!](\gamma^n)), [\![\boldsymbol{vmap}_n\{t\}]\!](\gamma')) \in R_\tau$$

The correctness of $\mathbf{vmap}_n\{-\}$ is obtained as a corollary of the fundamental lemma.

[2]Recall that a relation is functional if it specifies a function, i.e., for every x there is exactly one y such that $(x, y) \in R$.

Theorem 3.3 (Correctness of $\mathbf{vmap}_n\{-\}$). *If $\vdash t : T \to \tau$ is a closed program of function type, then for all v of type $T[n]$:*

- $\tau \equiv \eta$: $[\![\boldsymbol{vmap}_n\{t\}]\!](v) = \mathbf{zip}_\eta([\![t]\!](v[1]), \ldots, [\![t]\!](v[n]))$, *i.e.* $\boldsymbol{vmap}_n\{-\}$ *correctly vectorizes deterministic functions.*
- $\tau \equiv \boldsymbol{P}\,\eta$: $[\![\boldsymbol{vmap}_n\{t\}]\!](v) = \mathbf{zip}_{\eta_*}(\bigotimes_{i=1}^n [\![t]\!](v[i]))$, *i.e.* $\boldsymbol{vmap}_n\{-\}$ *produces vectors of independent samples when applied to stochastic functions.*
- $\tau \equiv \boldsymbol{G}_\gamma\ \eta$: $[\![\boldsymbol{vmap}_n\{t\}]\!](v) = (\mathbf{zip}_{\gamma_*}(\bigotimes_{i=1}^n [\![t]\!]_1(v[i])), \lambda r.\mathbf{zip}_\eta([\![t]\!]_2(v[1])(\mathbf{unzip}_\gamma(r)[1]), \ldots, [\![t]\!]_2(v[n])(\mathbf{unzip}_\gamma(r)[n]))$ *i.e. the trace distributions and return value maps of generative functions vectorized by* $\boldsymbol{vmap}_n\{-\}$ *are vectorizations of the original generative functions' trace distributions and return value maps.*

Here, $\mathbf{zip}_\eta : [\![\eta]\!]^n \to [\![\eta[n]]\!]$ *is the bijection between n-fold products of values in* $[\![\eta]\!]$ *and their (struct-of-array) vectorized representations* $[\![\eta[n]]\!]$, *with inverse* $\mathbf{unzip}_\eta : [\![\eta[n]]\!] \to [\![\eta]\!]^n$.

As an additional consequence of the fundamental lemma, we also get the following important commutativity relations, which we exploit in our implementation (Section 4.2):

Corollary 3.4. *Let $\vdash t : \boldsymbol{G}_\gamma\ \eta$ be a term of generative function type. Then:*

- $[\![\boldsymbol{simulate}\{\boldsymbol{vmap}_n\{t\}\}]\!] = \langle id, id, v \mapsto \prod_i v[i]\rangle_* [\![\boldsymbol{vmap}_n\{\boldsymbol{simulate}\{t\}\}]\!]$: *a correct implementation of* $\boldsymbol{simulate}\{\boldsymbol{vmap}_n\{t\}\}$ *can be obtained by applying* $\boldsymbol{vmap}_n\{-\}$ *to* $\boldsymbol{simulate}\{t\}$ *and collapsing the returned vector of densities to a single density via multiplication.*
- $[\![\boldsymbol{assess}\{\boldsymbol{vmap}_n\{t\}\}]\!] = \boldsymbol{let}\ a, b = [\![\boldsymbol{vmap}_n\{\boldsymbol{assess}\{t\}\}]\!]\ \boldsymbol{in}\ (a, \prod_i b[i])$: *a correct implementation of* $\boldsymbol{assess}\{\boldsymbol{vmap}_n\{t\}\}$ *can be obtained by applying* $\boldsymbol{vmap}_n\{-\}$ *to* $\boldsymbol{assess}\{t\}$ *and collapsing the returned vector of densities to a single density via multiplication.*

See the supplementary material for proofs.

3.5 Stochastic Branching

We can extend our formal model with support for stochastic branching, allowing us to account for vectorization of mixture models (including, e.g., the program in Fig. 6). To start, we add a construct for stochastic branching with homogeneous gradings, $\mathbf{cond}(t_1, t_2, t_3, t_4)$:

$$\frac{\Gamma \vdash t_1 : \mathbb{B}^s \quad \vdash t_2 : \eta_1 \to \mathbf{G}_\gamma\ \eta_2 \quad \vdash t_3 : \eta_1 \to \mathbf{G}_\gamma\ \eta_2 \quad \Gamma \vdash t_4 : \eta_1[s]}{\Gamma \vdash \mathbf{cond}(t_1, t_2, t_3, t_4) : \mathbf{G}_{\gamma[s]}\ \eta_2[s]}.$$

The expression $\mathbf{cond}(t_1, t_2, t_3, t_4)$, where t_1 has type $\mathbb{B}^s$, denotes the generative function that runs s-many independent executions of either t_2 or t_3, as selected by the provided Booleans in t_1, with arguments provided by t_4. Its trace distribution contains all s-many traces from these executions, and its return value function computes the s-many return values they yielded. We denote by $\mathbf{vmap}_{(n_1,\ldots,n_k)}$ the composition of program transformations $\mathbf{vmap}_{n_1} \circ \ldots \circ \mathbf{vmap}_{n_k}$ and by product : $\mathbb{R}[n] \to \mathbb{R}$ a new primitive construct which multiplies all its arguments. Now, we extend our program transformations to handle **cond** as follows. First, we define a useful lifting of **select** to operate component-wise on traces:

$$\begin{aligned}
&\mathbf{select}_{\{k_1:\eta_1,\ldots,k_m:\eta_m\}}(b, u, u') = \\
&\qquad \{k_1 : \mathbf{select}_{\eta_1}(b, u[k_1], u'[k_1]), \ldots, k_m : \mathbf{select}_{\eta_m}(b, u[k_m], u'[k_m])\}, \\
&\mathbf{select}_{T^s}(b, x, y) = \mathbf{select}(b, x, y),
\end{aligned}$$

with analogous clauses for products.

$$\begin{aligned}
&\textbf{simulate}\{\textbf{cond}(t_1, t_2, t_3, t_4)\} = \textbf{do}_{\mathbf{P}}\{\\
&\quad (u_2, v_2, w_2) \leftarrow \textbf{vmap}_s\{\textbf{simulate}\{t_2\}\}(t_4);\\
&\quad (u_3, v_3, w_3) \leftarrow \textbf{vmap}_s\{\textbf{simulate}\{t_3\}\}(t_4);\\
&\quad \textbf{let } u_{\text{sel}} = \textbf{select}_{\gamma[s]}(t_1, u_2, u_3);\\
&\quad \textbf{let } v_{\text{sel}} = \textbf{select}_{\eta_2[s]}(t_1, v_2, v_3);\\
&\quad \textbf{let } w_{\text{sel}} = \textbf{select}(t_1, w_2, w_3);\\
&\quad \textbf{return}_P\ (u_{\text{sel}}, v_{\text{sel}}, \text{product}(w_{\text{sel}}))\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{assess}\{\textbf{cond}(t_1, t_2, t_3, t_4)\} = \lambda u.\\
&\quad \textbf{let } (v_2, w_2) = \textbf{vmap}_s\{\textbf{assess}\{t_2\}\}(t_4)(u) \textbf{ in}\\
&\quad \textbf{let } (v_3, w_3) = \textbf{vmap}_s\{\textbf{assess}\{t_3\}\}(t_4)(u) \textbf{ in}\\
&\quad (\textbf{select}(t_1, v_2, v_3),\ \text{product}(\textbf{select}(t_1, w_2, w_3)))
\end{aligned}$$

$$\begin{aligned}
&\textbf{vmap}_n\{\textbf{cond}(t_1, t_2, t_3, t_4)\} =\\
&\textbf{cond}(\textbf{vmap}_n\{t_1\}, t_2, t_3, \textbf{vmap}_n\{t_4\})
\end{aligned}$$

Note that the use of $\textbf{vmap}_s$ within the **simulate**{} and **assess**{} transformations is sound because $\textbf{vmap}_s$ is only applied to closed terms. Our correctness results extend to this enriched language.

All examples in the paper use the homogeneous form above. In §4, we discuss how our implementation pads traces with sentinel values to support *heterogeneous* gradings. The supplementary material further extends our model to include generative primitives that internally use statically bounded loops via the **scan** function.

4 Implementation

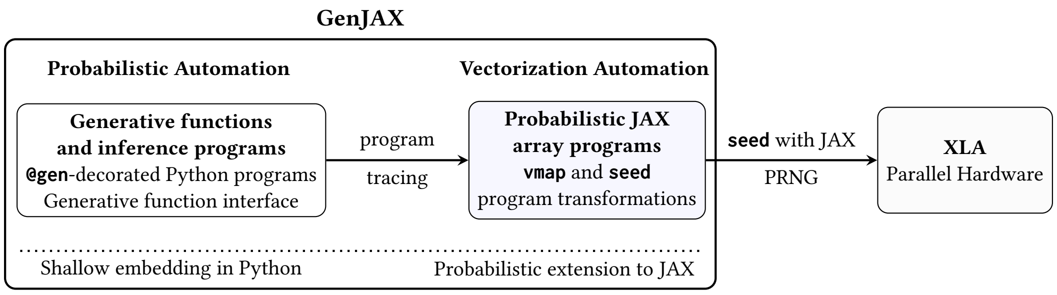

GenJAX
Probabilistic Automation
Vectorization Automation
Generative functions and inference programs
@gen-decorated Python programs
Generative function interface
program tracing
Probabilistic JAX array programs
vmap and seed program transformations
seed with JAX PRNG
XLA
Parallel Hardware
Shallow embedding in Python
Probabilistic extension to JAX

Fig. 15. GenJAX: compiler for vectorized programmable inference. Our compiler architecture extends support for JAX's vmap transformation to generative functions. Users express generative functions as @gen-decorated functions in Python. To support vmap, inference interfaces are transformed by GenJAX via program tracing into an intermediate representation that extends JAX with probabilistic sampling primitives. To lower and execute code, sampling primitives are eliminated by a seed transform, which allows GenJAX code to be executed by XLA on GPUs.

In this section, we present the key ideas behind the actual implementation of GenJAX atop JAX. An overview of our implementation is illustrated in Fig. 15. To expose an embedded Python DSL for our compiler, our implementation makes use of *lightweight effect handlers* [8, 69] to implement class methods corresponding to the program transformations presented in §3.1. When combined with JAX's support for program tracing (which performs partial evaluation on these lightweight effect handlers, thereby evaluating them away), this implementation strategy allows us to concisely embed our probabilistic interfaces and retain JAX compatibility.

4.1 Probabilistic Programming with Programmable Inference

GenJAX is implemented in Python, atop the JAX library for array programming [26].

Generative Functions. Generative functions are expressed by users as Python functions decorated with `@gen`. The bodies of generative functions may invoke deterministic JAX primitives for numerics and array programming, and draw string-named samples from primitive probability distributions exposed by our library (e.g. `x = normal(0, 1) @ "x"`). The `@gen` decorator wraps the user's program into a `GenerativeFunction` object, with methods corresponding to the methods of the generative function interface (`simulate`, `assess`, and others as described in Fig. 8).

Generative Function Interface Methods and Lightweight Effect Handling. When invoked, these methods use lightweight effect handling to intercept samples from primitive distributions and calls to other generative functions. Each method of the generative function interface defines its own effect handler class, and a fresh instance is pushed onto a global stack of active handlers when the method is invoked. The user's probabilistic program is then run. When a tracing expression is encountered (e.g., an expression of the form `normal(0,1) @ "x"`), control is transferred to the topmost handler on the stack. For example, when tracing an invocation of a primitive distribution, the handler object for `simulate` draws a sample from the specified distribution and records its value in a running trace (mirroring the definition in §3 of the behavior of `simulate` on the **trace** construct). Note that this strategy does not require delimited continuations or other heavy runtime features of full effect-handling systems.

Nonetheless, dynamic effect handling—and the use of Python data structures such as mutable dictionaries to accumulate traces—incurs some overhead. To eliminate this overhead, we rely on JAX's support for partial evaluation [27, 45, 62]. Given a generative function object, we run each method (`assess`, `simulate`, etc.) with symbolic inputs, and all computation is staged into a *Jaxpr*, a first-order array program in SSA form (see Fig. 16, right pane). At this point, all Python constructs, including those used to dispatch to effect handlers, have been partially evaluated away, leaving only JAX primitives. We extend JAX's built-in Jaxpr type to support two new primitive operations, `sample_p` for sampling a primitive distribution, and `log_dens_p` for evaluating the density of a primitive distribution.[3]

Traces. Traces are Python objects akin to dictionaries, which Jaxprs cannot directly manipulate. We use a JAX feature that allows us to register our Trace class as a `Pytree`, JAX's name for a nested Python container of arrays that can be flattened to and rebuilt from a list of arrays, by defining methods that convert traces to and from nested lists of arrays. These conversions are similar to those used to define the semantics of record types in our formal model: recall that although the syntax of a record type involves string-valued keys, our semantics maps every record type to a simpler nested-tuples-of-arrays representation. The Jaxprs we generate operate on such nested tuples of arrays.

4.2 Vectorization

Programmable Inference. Once a generative function method has been partially evaluated to a Jaxpr, we can vectorize the method by transforming the Jaxpr. All JAX primitives have built-in vectorized versions, analogous to the vectorized deterministic primitives in our formalization. We include special logic to vectorize the `sample_p` primitive, implementing the behavior of `vmap` on sample described in Section 3 (i.e., vectorized independent sampling).

Models / Generative Functions. To vectorize generative functions themselves (rather than just the inference programs that operate on generative functions), we rely on the commutativity result from §3 (Corollary 3.4). Generative function objects expose a `vmap` class method; calling

[3]`log_dens_p` is not strictly necessary, as log densities of individual primitive distributions can be implemented in terms of existing JAX primitives, but adding it makes the Jaxprs easier to read and debug.

it yields a new generative function object. Its `simulate` and `assess` methods are implemented by applying vectorization to the `simulate` and `assess` methods of the original generative function, and taking the product of the resulting vector of densities. Corollary 3.4 ensures that this is a correct implementation of these methods for the vmapped version of the original generative function.

Traces. Because traces are Pytree types, when JAX vectorizes a function which returns a trace, the returned vector of traces is represented as a nested-tuple-of-arrays. When a trace is constructed using the Pytree interface from the vectorized arrays, it is automatically in struct-of-array representation. This process, of identifying the "template" of a Pytree return value, using JAX to perform a computation, and then zipping arrays into the "template," is illustrated in Fig. 16 (**struct**).

4.3 Stochastic Branching

To support usage of generative functions with stochastic branching, we make use of JAX's built-in primitive `select_p`. Using the types of our formal model, the signature of `select_p` is **select_p** $:: T_{\mathbb{B}} \to T \to T \to T$ (where $T_{\mathbb{B}}$ is a batched Boolean). The behavior under evaluation corresponds to multiplexing of arrays using the Boolean input array as a selector.[4] Using `select_p`, we implement `cond`, which accepts two generative functions as inputs (corresponding to the branches of a conditional) and builds a generative function that branches between them. The branches are expected to accept the same types of arguments, and return the same type of value. `cond` can be invoked with the $\mathbb{B}[n]$ selector argument, and a vector of n arguments to the branches. The generative function interface methods are implemented as follows:

- For `simulate`, `cond` calls `simulate` on each of its branch generative functions. For returned densities, `select_p` is applied, using the selector argument. For traces, we reason about named addresses: if a name is used in both branches, the selector is used to select from the values of each branch. Otherwise, the selector is used to either pass through the value, or assign `NaN` (*not a number*)[5] to the address. This means that, unlike our formalism, our implementation supports the use of `cond` with branches that have heterogeneous trace types.
- For `assess`, `cond` calls `assess` on each of its branch generative functions. The same behavior as `simulate` above is used, for both densities and return values. Some of the computed densities will be `NaN`, but they will not be selected by `select_p`, so the product of selected densities will still be a number.

The implementations for `cond` of the other generative function interface methods (Fig. 8) follow the same strategies. Note that, in the variant described in our formal model (§3.5), the branches share a homogeneous graded trace type so that the selector can merge their traces component-wise. As mentioned above, our implementation supports *heterogeneous* grades by padding with `NaN` values. (The examples in this paper, however, do not rely on this generalization.)

4.4 Statically Bounded Loops

Our implementation also allows for a generative analogue of `jax.lax.scan` for looping generative functions. We provide a description of this extension in the supplementary material. The methods of the generative function interface are implemented by applying `jax.lax.scan` to the callee's implementation of the interface methods: the trace is batched by stacking the per-iteration traces, and densities are accumulated by combining the per-step contributions. The static bound restriction is important: the scan length N must be known during compilation because JAX requires static shapes to compile to XLA. Because probabilistic program traces reify the shape of the computation

[4]JAX does offer an explicit primitive for branching called `cond_p`, but under `vmap`, this primitive is automatically converted to `select_p`, so our implementation uses `select_p` directly.
[5]NaN is a special numeric value often used by numerical computing systems to represent undefined quantities.

into a recording of the execution, dynamically bounded loops entail traces whose shapes and sizes depend on runtime control flow – which directly violates JAX's requirements.

4.5 Execution on GPU

To compile the final Jaxpr to GPU, we first must eliminate our new primitives (**`sample_p`** and **`log_dens_p`**) so that the Jaxpr contains only standard (deterministic) JAX primitives. To do so, we replace **`sample_p`** operations with JAX's pseudorandom number generation primitives. These primitives operate on explicitly passed (splittable) counter-based random seeds [76]. Our **`seed`** transformation ensures that a random seed is split sufficiently many times for each vectorized call to use independent randomness for each dimension of its output.

Order of **`vmap`** *and* **`seed`**. Note that it is essential that seeding happens after vectorization. Using JAX's built-in **`vmap`** on JAX programs that are already written to use JAX's pseudorandom number generators would lead each vectorized random sampling operation to generate the *same* random number in every component of its output vector. By applying vectorization to a program with an explicit probabilistic primitive (**`sample_p`**), and *only then* introducing random seeds, our implementation can ensure seeds are appropriately split before they are passed as input to vectorized sampling operations.

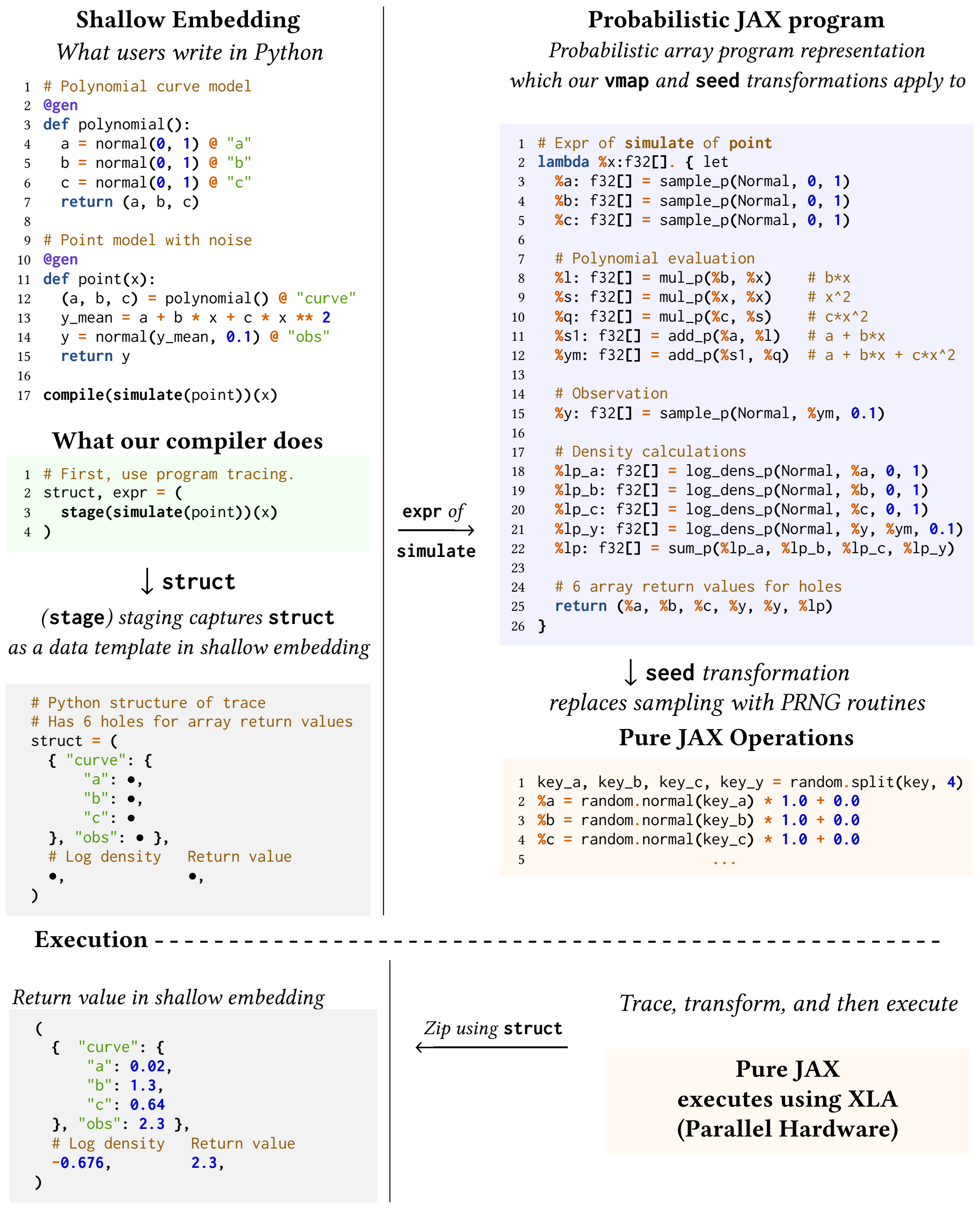
5 Evaluation

We evaluate our language and compiler implementation on benchmarks and case studies designed to assess the following criteria:

- (**Performance**) How does the performance of our compiler implementation compare to leading programmable inference systems? Do our abstractions introduce overhead compared to handcoded implementations of inference? *We survey the performance properties of GenJAX against open-source PPLs and tensor frameworks on standard modeling and inference tasks, for both embarrassingly-parallel algorithms (importance sampling) and iterative differentiable algorithms (Hamiltonian Monte Carlo).*
- (**Inference Quality**) **`vmap`** provides a convenient way to express inference problems over high-dimensional spaces. Does our design provide the means to construct effective inference approximations for them? *We study probabilistic Game of Life inversion on large boards using approximate inference, and use GenJAX to construct an efficient nested vectorized Gibbs sampler. We study a probabilistic model of robot localization using simulated LIDAR measurements, and use GenJAX to iteratively construct sequential Monte Carlo [20, 23] (SMC) algorithms, including an efficient algorithm using proposals with vectorized locally optimal grid approximations.*

5.1 Performance Survey Evaluation

Figure 17 presents a performance survey of our system compared to open-source tensor frameworks and PPLs across a handful of models and inference algorithms. In the top panel, we examine the runtime characteristics of our compiler on importance sampling in a Beta-Bernoulli model. The model infers the bias of a coin from observed flips, using a Beta(1,1) prior and Bernoulli likelihood. We observe 50 flips, and construct a posterior approximation using importance sampling. The top panel confirms that all frameworks accurately recover the true posterior distribution. GenJAX achieves near-identical performance to handcoded JAX (100.1% relative time). The bottom panel of Figure 17 presents performance results for importance sampling and Hamiltonian Monte Carlo (HMC) [67] on the polynomial regression problem from §2. Importance sampling exhibits parallel scaling with the number of particles: vectorized PPLs and tensor frameworks have near constant scaling while the GPU is not saturated. HMC is run iteratively: here, the scaling is linear in the



Shallow Embedding

What users write in Python

```
# Polynomial curve model
@gen
def polynomial():
  a = normal(0, 1) @ "a"
  b = normal(0, 1) @ "b"
  c = normal(0, 1) @ "c"
  return (a, b, c)

# Point model with noise
@gen
def point(x):
  (a, b, c) = polynomial() @ "curve"
  y_mean = a + b * x + c * x ** 2
  y = normal(y_mean, 0.1) @ "obs"
  return y

compile(simulate(point))(x)
```

What our compiler does

```
# First, use program tracing.
struct, expr = (
  stage(simulate(point))(x)
)
```

↓ **struct**

(**stage**) *staging captures* **struct** *as a data template in shallow embedding*

```
# Python structure of trace
# Has 6 holes for array return values
struct = (
  { "curve": {
      "a": •,
      "b": •,
      "c": •
  }, "obs": • },
  # Log density   Return value
  •,              •,
)
```

expr *of* → **simulate**

Probabilistic JAX program

Probabilistic array program representation which our **vmap** *and* **seed** *transformations apply to*

```
# Expr of simulate of point
lambda %x:f32[]. { let
  %a: f32[] = sample_p(Normal, 0, 1)
  %b: f32[] = sample_p(Normal, 0, 1)
  %c: f32[] = sample_p(Normal, 0, 1)

  # Polynomial evaluation
  %l: f32[] = mul_p(%b, %x)     # b*x
  %s: f32[] = mul_p(%x, %x)     # x^2
  %q: f32[] = mul_p(%c, %s)     # c*x^2
  %s1: f32[] = add_p(%a, %l)    # a + b*x
  %ym: f32[] = add_p(%s1, %q)   # a + b*x + c*x^2

  # Observation
  %y: f32[] = sample_p(Normal, %ym, 0.1)

  # Density calculations
  %lp_a: f32[] = log_dens_p(Normal, %a, 0, 1)
  %lp_b: f32[] = log_dens_p(Normal, %b, 0, 1)
  %lp_c: f32[] = log_dens_p(Normal, %c, 0, 1)
  %lp_y: f32[] = log_dens_p(Normal, %y, %ym, 0.1)
  %lp: f32[] = sum_p(%lp_a, %lp_b, %lp_c, %lp_y)

  # 6 array return values for holes
  return (%a, %b, %c, %y, %y, %lp)
}
```

↓ **seed** *transformation replaces sampling with PRNG routines*

Pure JAX Operations

```
key_a, key_b, key_c, key_y = random.split(key, 4)
%a = random.normal(key_a) * 1.0 + 0.0
%b = random.normal(key_b) * 1.0 + 0.0
%c = random.normal(key_c) * 1.0 + 0.0
                    ...
```

Execution

Return value in shallow embedding

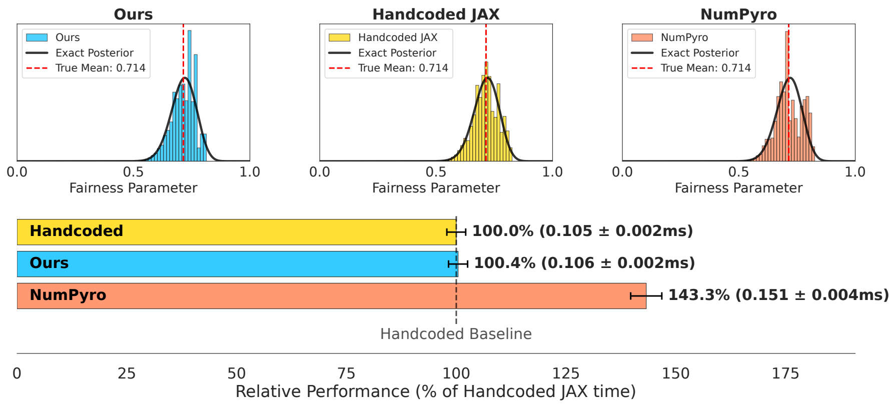
```
(
  {  "curve": {
       "a": 0.02,
       "b": 1.3,
       "c": 0.64
  }, "obs": 2.3 },
  # Log density    Return value
  -0.676,          2.3,
)
```

← *Zip using* **struct**

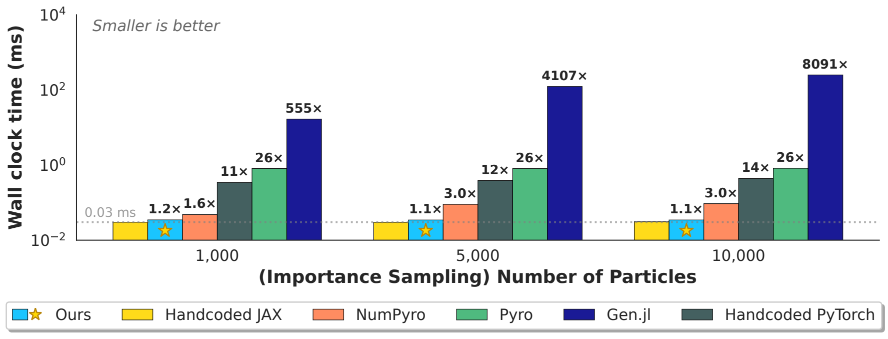
Trace, transform, and then execute

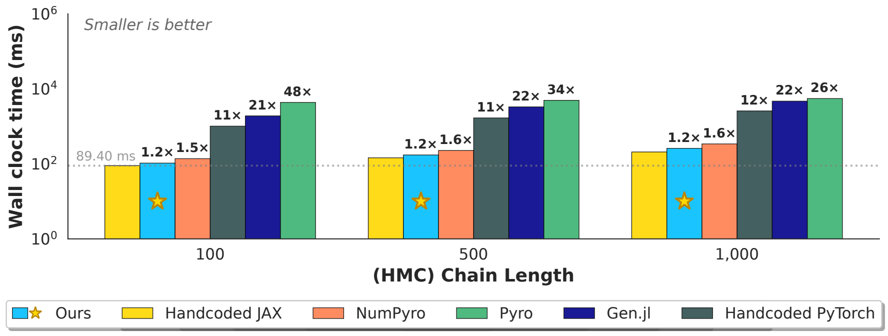
Pure JAX executes using XLA (Parallel Hardware)

Fig. 16. How our compiler works. (Left, top) Users write high-level probabilistic programs in Python. Interfaces (`simulate`) use lightweight effect handlers to intercept `@` operations during execution. Our compiler uses program tracing (`stage`) to transform the implementation into an array program intermediate representation with probabilistic primitives (right, top), and captures static host-language structure for the return type (left, middle). (Right, bottom) The seed transformation eliminates probabilistic primitives for explicit pseudorandom samplers, producing pure JAX operations for hardware execution. The result is executed via XLA, and returned to our shallow embedding into the host-language structure.



(a) Beta-Bernoulli inference accuracy and timing comparison. Comparing the overhead of inference approximations constructed via importance sampling using GenJAX's abstractions to handcoded JAX programs and NumPyro.





(b) Polynomial regression survey. Comparing wall clock runtimes for importance sampling and Hamiltonian Monte Carlo on polynomial regression (§2).

Fig. 17. Performance evaluation across probabilistic programming frameworks. (a) Beta-Bernoulli inference comparing posterior accuracy and execution time for GenJAX, NumPyro, and handcoded JAX implementations with 50 observations and 2000 samples, demonstrating importance sampling using GenJAX's programmable inference abstractions is competitive with handcoded performance. (b) Scaling analysis across six frameworks showing GenJAX achieves performance is consistently near handcoded JAX and competitive with other open-source PPLs, for both importance sampling and Hamiltonian Monte Carlo.

length of the chain. GenJAX is consistently close to handcoded and optimized JAX, validating that our abstractions for programmable inference introduce minimal overhead.

5.2 High-Dimensional Vectorized Inference

In this section, we illustrate the usage of GenJAX to develop two more involved applications of modeling and inference: probabilistic inversion of Conway's *Game of Life* [28] (a massive discrete search problem over cellular automaton dynamics) and 2D robot localization (a subproblem in simultaneous localization and mapping [24]).

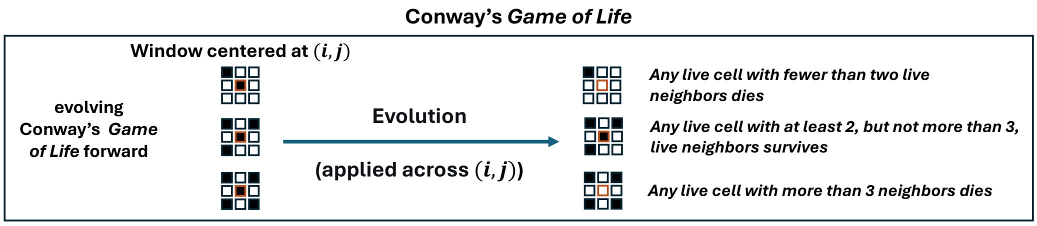

Conway's *Game of Life*
Window centered at (i, j)
evolving Conway's *Game of Life* forward
Evolution
(applied across (i, j))
Any live cell with fewer than two live neighbors dies
Any live cell with at least 2, but not more than 3, live neighbors survives
Any live cell with more than 3 neighbors dies

Fig. 18. Deterministic evolution rules for Conway's *Game of Life*. In our case study, we add a uniform prior over board state, and probabilistic Bernoulli noise on top of the deterministic rules to construct a *Game of Life* generative function. We can then condition the observed next state, and construct an inference problem whose solutions correspond to approximate inversions of the Game of Life dynamics.

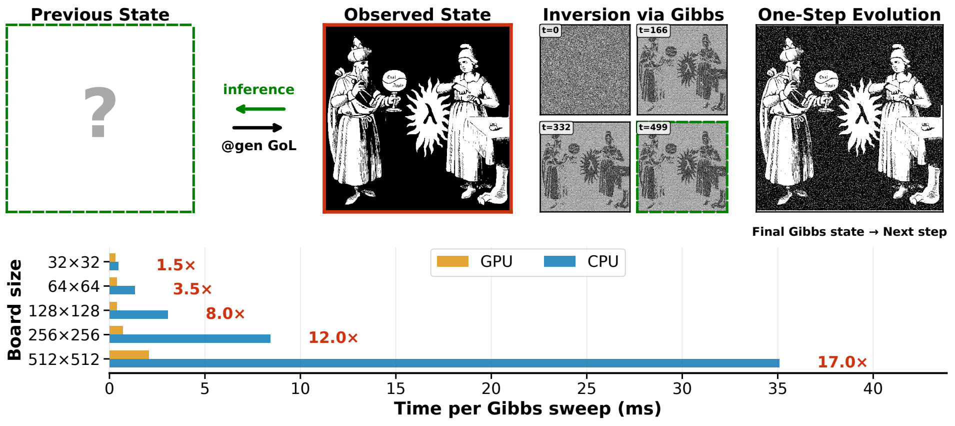


Fig. 19. Vectorized Gibbs sampling in the *Game of Life*. Probabilistic Game of Life inversion on the wizard book cover [2] (1024 × 1024 grid). Top: (1) Previous state (unknown, the target of our inference process); (2) Observed future state (the target pattern); (3) Vectorized Gibbs chain showing states constructed by inference in a progression from $t = 0$ to $t = 499$; (4) One-step deterministic evolution of final inferred state, reconstruction accuracy (measured as discrepancy between bits) is around 90%. Bottom: Benchmark timings of single vectorized Gibbs sweep performance across board sizes, comparing CPU and GPU execution times. GPU execution timings demonstrate the benefit of parallel hardware acceleration for vectorized inference. Overall: the runtime takes about 2.8 seconds for 500 iterations on an RTX 4090 GPU, with about 93% reconstruction accuracy (70,109 bits out of 1,048,576 total bits).

Probabilistic Game of Life Inversion. Game of Life (GoL) inversion is the problem of inverting the dynamics of Conway's *Game of Life* [28]: given a final state, what is a possible previous state that evolves to the final state under the rules of the game (Fig. 18)? Brute force discrete search is computationally intractable, requiring evaluation of $2^{N \times N}$ states, where N is the linear dimension of a square GoL game board. In this case study, we introduce probabilistic noise into the dynamics of GoL: from an initial state, we evolve forward using the deterministic rules, but then sample with Bernoulli noise around the true value of the state of each pixel (i.e., the observed value of a pixel has a small chance of being opposite the true value). In Fig. 19, we illustrate approximate inversion using vectorized Gibbs sampling [29]. Because each cell's value is conditionally independent from non-neighboring cells' values, given its eight neighbors, we partition the board's cells into conditionally independent groups (given the other groups). Within each group, we can perform parallel Gibbs updates on all the cells, an example of chromatic Gibbs [31]. The generative function representing probabilistic GoL dynamics and the vectorized Gibbs algorithm are all written using GenJAX's abstractions. The result is a highly efficient probabilistic inversion algorithm which can invert Life states with up to 90% accuracy in a few seconds.

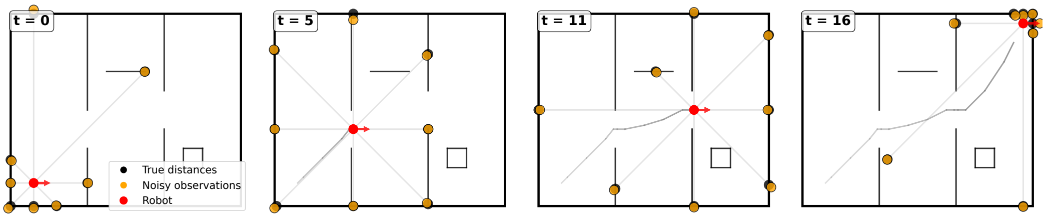
Robot Localization. In robotics, simultaneous mapping and localization (SLAM) refers to the problem of constructing a representation of the map of an environment and the position of the robot within the map based on measurements (often, LIDAR-based measurements). If the map is given, the problem is called localization (Fig. 20a): a robot maneuvers through a known space, and receives measurements of distance to the walls. The goal is to construct a probabilistic representation of where the robot is located. In this case study, we use GenJAX to write a model for robot localization, with Gaussian drift dynamics and a simulated LIDAR measurement. Given a sequence of LIDAR measurements over time as observations, we can then constrain the model to produce a posterior over robot locations. In Fig. 20b, we develop several sequential Monte Carlo algorithms using GenJAX's programmable inference abstractions.

- The **bootstrap filter** [34] is sequential Monte Carlo where the prior (from the model) is used as the proposal for the latent position of the robot.
- **SMC + HMC** adds HMC [67] moves to the bootstrap filter. These moves are applied to the particle collection after resampling.
- **SMC + Locally Optimal** uses a smart proposal for the latent position of the robot based on enumerative grids: the logic of the proposal is to enumerate a grid in position space, and evaluate each position on the grid against the observation likelihood. The maximum likelihood grid point is selected, and then a proposal for the position is sampled from a normal distribution around that point.

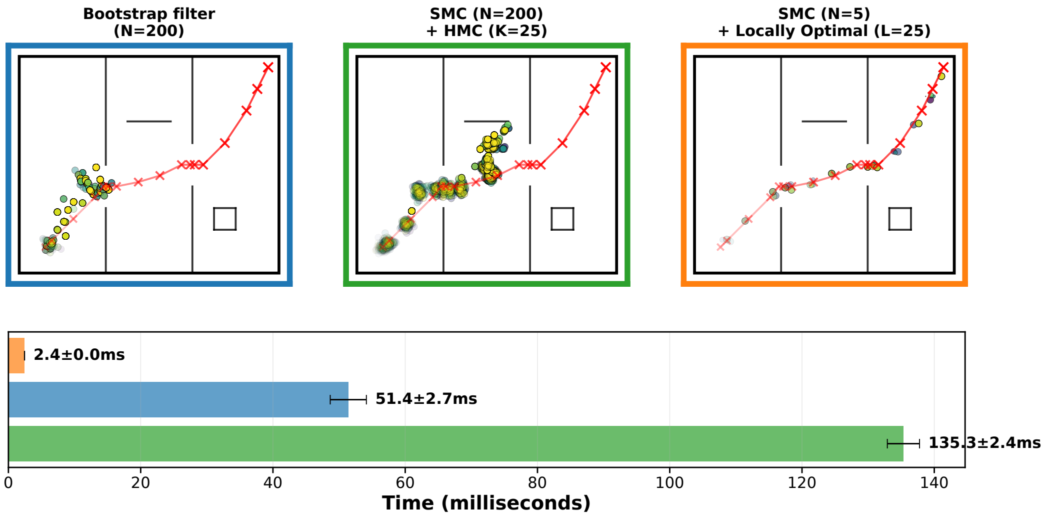
SMC supports natural vectorization over particles. In our experiments, from the standpoint of efficiency and accuracy, the best algorithm is locally optimal SMC, which adds another layer of vectorization *within* the custom proposal. In the locally optimal grid approximation proposal, the likelihood grid evaluations can be fully vectorized. Note that the model already features vectorization: the LIDAR measurement model is vectorized as well. The development of this algorithm illustrates the power of exposing `vmap` as an idiom: each of these opportunities for vectorization (in the model, in the locally optimal proposal, and across the particle collection) are convenient to program against with `vmap`, and lead to a highly efficient inference algorithm which can accurately track the 2D robot's location within the map in milliseconds.

6 Related Work

Probabilistic Programming. Early probabilistic languages such as Church [32], WebPPL [33], Venture [60, 61], and Anglican [82] prioritized modeling expressiveness over GPU inference. Domain-specific probabilistic programming languages with GPU backends include Augur [85], which



(a) The robot localization problem. Given a fixed map of the area, a robot must localize its location using simulated LIDAR measurements: rays are cast out from the robot's location, and a noisy measurements of the distance to any intersecting object are returned. Given these measurements and the known map, the goal is to construct a probability distribution over the location of the robot.



(b) Comparison between SMC variants. SMC can be customized in various ways: custom proposals and MCMC moves can be used to improve the accuracy of the algorithm. The best algorithm (SMC + Locally Optimal) uses a custom proposal which uses vmap to evaluate positions on a grid using the data likelihood, and then samples from a normal around the most promising grid point. Note that this custom inference program is faster and more accurate (orange) than naive scale-up of a vectorized bootstrap filter (blue), and faster and more accurate than a hybrid of SMC with Hamiltonian Monte Carlo rejuvenation (green).

Fig. 20. Robot localization using programmable sequential Monte Carlo. (a) Problem setup showing robot trajectory through multi-room environment with 8-ray LIDAR sensor model for distance-based localization. (b) Comparison of three SMC variants: Bootstrap filter, SMC+HMC, and SMC+Locally Optimal, showing particle approximation evolution and execution time performance.

compiles Bayesian networks to data-parallel code; RootPPL [59], which targets CUDA for phylogenetic inference; Birch [65], which supports delayed sampling for particle filters and CUDA execution; and Stan [14], which restricts models to fixed control flow for HMC.

GenJAX is an embedded probabilistic programming framework, leveraging JAX for differentiable computation and JIT compilation for good performance on GPUs. In this respect, GenJAX is similar to Edward/Edward2 [83, 84], PyMC3 [77], Pyro [8], and NumPyro [69], which respectively leverage TensorFlow, PyTorch, and JAX to execute vectorized inference code on GPUs. Inference families including Monte Carlo methods (for instance, Hamiltonian Monte Carlo [67]) and variational inference methods [9, 46, 48, 71] often feature embarrassingly parallel subroutines, and benefit from vectorization and GPU acceleration.

Compared to Pyro and NumPyro, GenJAX's design is aimed at improving expressivity while retaining vectorization opportunities. GenJAX's `cond` construct is an example of this philosophy: branches to `cond` are restricted to be valid GenJAX generative functions, which are less expressive than Pyro's modeling language. However, `cond` supports stochastic branching and is fully compatible with `vmap`. In contrast, neither NumPyro nor Pyro support a stochastic branching primitive. Instead, users of these systems emulate stochastic branching manually, by inlining branches into model code, and using masking operations on distributions. Also, Pyro's `plate` construct for model vectorization is implemented as an effect handler that alters the meaning of array operations. In contrast, by using an approach based on program transformation, GenJAX allows users to freely nest vectorization of models and inference algorithms, and use ordinary JAX constructs within models for stochastic branching. GenJAX also supports vectorized implementations of generative functions [19], and custom inference programs that interleave automated and hand-optimized implementations of the generative function interface. The benchmarks in this paper show GenJAX delivers these capabilities while maintaining competitive performance to NumPyro and introducing low overhead relative to hand-optimized JAX. The programming model for users of GenJAX is similar to Pyro and NumPyro: users are required to make use of `vmap` in their code to benefit from vectorization. Recent work [55] explores automating vectorization of sequential data-dependent loops without requiring user annotation using speculative execution, iterative correction, and fixed-point checks: this work could be integrated into GenJAX via a new type of generative function whose internal logic uses `vmap` to implement this vectorization technique.

Data-Parallel and Array Programming. Our work builds directly on JAX [26], which itself builds on foundational languages for data-parallel array programming. NESL [10] introduced the idea that nested parallelism can make use of program transformations: flattening converts nested operations like `{sum(a) : a in arrays}` into operations on flat vectors, an idea reincarnated into JAX in the form of JAX's `Pytree` interface. APL [43] and J [40] introduced rank polymorphism to array programming: operations work uniformly across dimensions, which has been translated into axis-specified reduction primitives in NumPy [37] and JAX. Modern array programming frameworks in Python (NumPy [37], TensorFlow [1], PyTorch [68], JAX [26]) have brought significant attention to array programming from data science and artificial intelligence research. JAX's `vmap` inherits NESL's transformation-based parallelism but focuses on deterministic computation.

Partial Evaluation and Staged Computation. Partial evaluation [27, 44, 45, 62] and multi-stage programming [80] have played a significant role in the development of techniques for *program tracing*, which JAX relies upon to support program transformation and compositional interpreters as program transformers. GenJAX relies upon JAX's support for program tracing to extend `vmap` to work on probabilistic constructs. Additionally, JAX's program tracing is used to eliminate the overhead of GenJAX's lightweight effect handler implementations of the generative function interface. Several PPLs have made use of partial evaluation to improve the accuracy or runtime of inference: Hakaru [66] can partially evaluate a subset of its programs to closed form when possible, Gen [19] supports trace data structure specialization on the structure of generative functions in its static modeling language.

Formalization of Sound Bayesian Inference. Vectorization poses interesting questions for soundness: operations which are correct pointwise should preserve measure-theoretic properties when lifted to operate on arrays. In our formal model and soundness results, we rely on the quasi-Borel space framework [39] to construct our denotational semantics. Several works influenced our development: Borgström et al. [11] establishes lambda-calculus foundations for universal PPLs; Ščibior et al. [78] validates that transformations preserve Bayesian soundness. Our system extends **`vmap`** to work with probabilistic program traces and programmable inference interfaces, and we rely on several prior semantic developments, including Lew et al. [50], which introduced trace typing for ensuring proposal-model alignment in programmable inference algorithms, and Lew et al. [54], which developed a denotational semantics model for programmable inference with traces, using a graded monad similar to ours to track a trace's shape. We build on these works by giving a model for how vectorization interacts with programmable inference features like tracing. Following works like [7, 42, 52, 54], our formal developments also rely on logical relations arguments to reason about the correctness of probabilistic program transformations.

7 Conclusion

This work presents GenJAX, a language and compiler for vectorized probabilistic programming with programmable inference. This system integrates **`vmap`** with programmable inference features: we extend **`vmap`** support to generative functions, including support for vectorization using **`vmap`** of probabilistic program traces, stochastic branching, and programmable inference interfaces. Benchmarks show this approach yields low overhead relative to hand-optimized JAX, and simultaneously delivers greater expressiveness and competitive performance with other probabilistic programming systems targeting modern accelerators.

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

- **Vectorized inference diagnostics.** By automating the vectorized implementation of nested models and inference algorithms, GenJAX makes it easy to experiment with parallel implementations of custom Monte Carlo estimators of a broad range of information-theoretic quantities derived from probabilistic programs [18, 22, 73, 75], including KL divergence between inference algorithms and the conditional mutual information among subsets of latent variables. Although computationally intensive on CPUs, these estimators are comprised of nested, massively parallel computations, and may become more practical and widespread given suitable automation.
- **Spatial or geometric probabilistic programs.** We expect that GenJAX's support for array programming and programmable probabilistic inference may be well-suited for spatial computing applications. Domains such as robotics, autonomous navigation, computational imaging, and scientific simulation increasingly require sophisticated probabilistic reasoning over high-dimensional spatial data—including LiDAR point clouds, depth images, and other spatial data types. Probabilistic programming applications in these domains naturally involve computations that manipulate multi-dimensional arrays. GenJAX's design is uniquely suited to support practitioners writing these types of probabilistic programs, and provides useful vectorization automation and support for compilation to efficient GPU implementations.

Data-Availability Statement

The artifact associated with this paper is available on Zenodo [6]. The source code is available at https://github.com/probcomp/genjax [63].

Acknowledgments

The authors acknowledge Arijit Dasgupta, Andrew Bolton, João Loula, Nishad Gothoskar, Matin Ghavami, Eric Li, Ian Limarta, Jay Pottharst, Jack Rusher, Matt Huebert, David R. MacIver, Mirko Klukas, Fabian Zaiser, Ben Lee, Brian Patton, Jacob Burnhim, Christopher Suter, Urs Köster, Nathan Cloos, Pierre Glaser, and Alex Hiser for assisting in the development of our system. We acknowledge

Sharad Vikram and Matthew Johnson for useful discussions about JAX. We acknowledge Sam Witty, Tan Zhi-Xuan, Timothy O'Donnell, and Josh Tenenbaum for helpful discussions concerning our work. We acknowledge Karen Laska Pierro for support during preparation of this work. This work was supported in part by CoCoSys, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. doi:10.48550/arXiv.1603.04467 Software available from tensorflow.org.

[2] Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, Cambridge, MA, USA.

[3] Alwa Alanqary, Gloria Z. Lin, Joie Le, Tan Zhi-Xuan, Vikash Mansinghka, and Josh Tenenbaum. 2021. Modeling the Mistakes of Boundedly Rational Agents Within a Bayesian Theory of Mind. *Proceedings of the Annual Meeting of the Cognitive Science Society* 43, 43 (2021). doi:10.48550/arXiv.2106.13249

[4] Chris L. Baker, Rebecca R. Saxe, and Joshua B. Tenenbaum. 2011. Bayesian Theory of Mind: Modeling Joint Belief-Desire Attribution. In *Proceedings of the 33rd Annual Conference of the Cognitive Science Society.* Cognitive Science Society, Boston, MA, USA, 2469–2474. https://escholarship.org/uc/item/5rk7z59q

[5] Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19).* Association for Computing Machinery, New York, NY, USA, 1–24. doi:10.1145/3295500.3356180

[6] McCoy R. Becker, Mathieu Huot, George Matheos, Xiaoyan Wang, Karen Chung, Colin Smith, Sam Ritchie, Alexander K. Lew, Martin Rinard, and Vikash K. Mansinghka. 2025. *GenJAX: Probabilistic Programming with Vectorized Programmable Inference.* doi:10.5281/zenodo.17594132

[7] McCoy R. Becker, Alexander K. Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C. Rinard, and Vikash K. Mansinghka. 2024. Probabilistic Programming with Programmable Variational Inference. *Proc. ACM Program. Lang.* 8, PLDI (June 2024), 2123–2147. doi:10.1145/3656463

[8] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2018), 1–6. doi:10.48550/arXiv.1810.09538

[9] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (2017), 859–877. doi:10.1080/01621459.2017.1285773

[10] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. *J. Parallel and Distrib. Comput.* 21, 1 (1993), 4–14. doi:10.1006/jpdc.1994.1031

[11] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming.* ACM, New York, NY, USA, 33–46. doi:10.1145/2951913.2951942

[12] Monica F. Bugallo, Victor Elvira, Luca Martino, David Luengo, Joaquin Miguez, and Petar M. Djuric. 2017. Adaptive Importance Sampling: The past, the present, and the future. *IEEE Signal Processing Magazine* 34, 4 (July 2017), 60–79. doi:10.1109/MSP.2017.2699226

[13] Olivier Cappé, Randal Douc, Arnaud Guillin, Jean-Michel Marin, and Christian P. Robert. 2008. Adaptive importance sampling in general mixture classes. *Statistics and Computing* 18, 4 (Dec. 2008), 447–459. doi:10.1007/s11222-008-9059-x

[14] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32. doi:10.18637/jss.v076.i01

[15] Katherine M. Collins, Ilia Sucholutsky, Umang Bhatt, Kartik Chandra, Lionel Wong, Mina Lee, Cedegao E. Zhang, Tan Zhi-Xuan, Mark Ho, Vikash Mansinghka, Adrian Weller, Joshua B. Tenenbaum, and Thomas L. Griffiths. 2024. Building Machines that Learn and Think with People. doi:10.48550/arXiv.2408.03943 arXiv:2408.03943 [cs].

[16] Aidan Curtis, George Matheos, Nishad Gothoskar, Vikash Mansinghka, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. 2024. Partially Observable Task and Motion Planning with Uncertainty and Risk Awareness.

In *Robotics: Science and Systems XX, Delft, The Netherlands, July 15-19, 2024*, Dana Kulic, Gentiane Venture, Kostas E. Bekris, and Enrique Coronado (Eds.). Robotics: Science and Systems Foundation, Delft, The Netherlands, 118:1–118:9. doi:10.15607/RSS.2024.XX.118

[17] Marco F. Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. 2020. Automating Involutive MCMC using Probabilistic and Differentiable Programming. *CoRR* abs/2007.09871 (July 2020). doi:10.48550/arXiv.2007.09871 arXiv:2007.09871 [cs.LG]

[18] Marco F. Cusumano-Towner and Vikash K. Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. *CoRR* abs/1705.07224 (Nov. 2017). doi:10.48550/arXiv.1705.07224 arXiv:1705.07224 [cs.AI]

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

[20] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (June 2006), 411–436. doi:10.1111/j.1467-9868.2006.00553.x

[21] A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39, 1 (Sept. 1977), 1–22. doi:10.1111/j.2517-6161.1977.tb01600.x

[22] Justin Domke. 2021. An Easy to Interpret Diagnostic for Approximate Inference: Symmetric Divergence Over Simulations. *CoRR* abs/2103.01030 (Feb. 2021). doi:10.48550/arXiv.2103.01030

[23] Arnaud Doucet, Nando de Freitas, and Neil Gordon (Eds.). 2001. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, New York. doi:10.1007/978-1-4757-3437-9

[24] Hugh Durrant-Whyte and Tim Bailey. 2006. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation Magazine* 13, 2 (2006), 99–110. doi:10.1109/MRA.2006.1638022

[25] Shai Fine, Yoram Singer, and Naftali Tishby. 1998. The Hierarchical Hidden Markov Model: Analysis and Applications. *Machine Learning* 32, 1 (July 1998), 41–62. doi:10.1023/A:1007469218079

[26] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. SysML 2018.

[27] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation* 12, 4 (Dec. 1999), 377–380. doi:10.1023/A:1010043619517

[28] Martin Gardner. 1970. The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American* 223, 4 (1970), 120–123. doi:10.1038/scientificamerican1070-120

[29] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6, 6 (1984), 721–741. doi:10.1109/TPAMI.1984.4767596

[30] Walter R. Gilks and Carlo Berzuini. 2001. Following a Moving Target: Monte Carlo Inference for Dynamic Bayesian Models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 1 (2001), 127–146. doi:10.1111/1467-9868.00280

[31] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. 2011. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *AISTATS (JMLR Proceedings, Vol. 15)*. JMLR.org, Fort Lauderdale, FL, USA, 324–332. https://proceedings.mlr.press/v15/gonzalez11a.html

[32] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*. AUAI Press, Corvallis, OR, USA, 220–229. doi:10.48550/arXiv.1206.3255

[33] Noah D. Goodman and Andreas Stuhlm"uller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Electronic; retrieved 2025-07-13.

[34] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)* 140, 2 (1993), 107–113. doi:10.1049/ip-f-2.1993.0015

[35] Nishad Gothoskar, Marco Cusumano-Towner, Ben Zinberg, Matin Ghavamizadeh, Falk Pollok, Austin Garrett, Joshua B. Tenenbaum, Dan Gutfreund, and Vikash K. Mansinghka. 2021. 3DP3: 3D Scene Perception via Probabilistic Programming. In *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, Vol. 34. Curran Associates, Inc., Virtual. doi:10.48550/arXiv.2111.00312

[36] Peter J. Green. 1995. Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika* 82, 4 (1995), 711–732. doi:10.2307/2337340 Publisher: [Oxford University Press, Biometrika Trust].

[37] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. doi:10.1038/s41586-020-2649-2

[38] W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109. doi:10.2307/2334940 Publisher: [Oxford University Press, Biometrika Trust].

[39] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, Reykjavik, Iceland, 1–12. doi:10.1109/LICS.2017.8005137

[40] Roger K.W. Hui, Kenneth E. Iverson, Eugene E. McDonnell, and Arthur T. Whitney. 1990. APL? *APL Quote Quad* 20, 4 (1990). http://www.jsoftware.com/papers/J1990.htm Proceedings of APL90, Copenhagen.

[41] Mathieu Huot, Matin Ghavami, Alexander K. Lew, Ulrich Schaechtle, Cameron E. Freer, Zane Shelby, Martin C. Rinard, Feras A. Saad, and Vikash K. Mansinghka. 2024. GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables. *Proc. ACM Program. Lang.* 8, PLDI (June 2024), 790–815. doi:10.1145/3656409

[42] Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of automatic differentiation via diffeologies and categorical gluing. In *Foundations of Software Science and Computation Structures (FoSSaCS 2020) (Lecture Notes in Computer Science, Vol. 12077)*. Springer, Cham, Switzerland, 319–338. doi:10.1007/978-3-030-45231-5_17

[43] Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley and Sons, New York, NY, USA.

[44] Neil D. Jones. 1996. An introduction to partial evaluation. *Comput. Surveys* 28, 3 (Sept. 1996), 480–503. doi:10.1145/243439.243447

[45] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall, Englewood Cliffs, NJ, USA.

[46] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. 1999. An Introduction to Variational Methods for Graphical Models. *Machine Learning* 37, 2 (1999), 183–233. doi:10.1023/A:1007665907178

[47] Chang-Jin Kim. 1994. Dynamic linear models with Markov-switching. *Journal of Econometrics* 60, 1 (Jan. 1994), 1–22. doi:10.1016/0304-4076(94)90036-1

[48] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. doi:10.48550/arXiv.1312.6114

[49] Alexander K. Lew, Monica Agrawal, David Sontag, and Vikash K. Mansinghka. 2020. PClean: Bayesian Data Cleaning at Scale with Domain-Specific Probabilistic Programming. In *International Conference on Artificial Intelligence and Statistics*. PMLR, Virtual Event, 1927–1935. doi:10.48550/arXiv.2007.11838

[50] Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proc. ACM Program. Lang.* 4, POPL (2019), 19:1–19:32. doi:10.1145/3371087

[51] Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. 2023. Probabilistic Programming with Stochastic Probabilities. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1708–1732. doi:10.1145/3591290

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

[53] Alexander K. Lew, George Matheos, Tan Zhi-Xuan, Matin Ghavamizadeh, Nishad Gothoskar, Stuart Russell, and Vikash K. Mansinghka. 2023. SMCP3: Sequential Monte Carlo with Probabilistic Program Proposals. In *International Conference on Artificial Intelligence and Statistics, 25-27 April 2023, Palau de Congressos, Valencia, Spain (Proceedings of Machine Learning Research, Vol. 206)*, Francisco J. R. Ruiz, Jennifer G. Dy, and Jan-Willem van de Meent (Eds.). PMLR, Valencia, Spain, 7061–7088. https://proceedings.mlr.press/v206/lew23a.html

[54] Alexander K. Lew, Eli Sennesh, Jan-Willem Van De Meent, and Vikash K. Mansinghka. 2023. Semantics of Probabilistic Program Traces. In *LAFI 2023 at POPL* (Boston, MA). ACM, Boston, MA, USA. https://popl23.sigplan.org/details/lafi-2023-papers/1/Semantics-of-Probabilistic-Program-Traces

[55] Sangho Lim, Hyoungjin Lim, Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2025. Optimising Density Computations in Probabilistic Programs via Automatic Loop Vectorisation. doi:10.48550/arXiv.2511.11070 arXiv:2511.11070 [cs.PL]

[56] Scott Linderman, Matthew Johnson, Andrew Miller, Ryan Adams, David Blei, and Liam Paninski. 2017. Bayesian Learning and Inference in Recurrent Switching Linear Dynamical Systems. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. PMLR, Fort Lauderdale, FL, USA, 914–922. https://proceedings.mlr.press/v54/linderman17a.html ISSN: 2640-3498.

[57] Benjamin Lipkin, Benjamin LeBrun, Jacob Hoover Vigly, João Loula, David R. MacIver, Li Du, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Timothy J. O'Donnell, Alexander K. Lew, and Tim Vieira. 2025. Fast Controlled Generation from Language Models with Adaptive Weighted Rejection Sampling. doi:10.48550/arXiv.2504.05410 arXiv:2504.05410 [cs].

[58] João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. 2025. Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo. *CoRR* abs/2504.13139

(2025). doi:10.48550/arXiv.2504.13139 arXiv:2504.13139

[59] Daniel Lundén, Joey Öhman, Jan Kudlicka, Viktor Senderov, Fredrik Ronquist, and David Broman. 2022. Compiling Universal Probabilistic Programming Languages with Efficient Parallel Sequential Monte Carlo Inference. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, Munich, Germany, 29–56. doi:10.1007/978-3-030-99336-8_2

[60] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014). doi:10.48550/arXiv.1404.0099 arXiv:1404.0099 [cs.AI]

[61] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. *ACM SIGPLAN Notices* 53, 4 (2018), 603–616. doi:10.1145/3296979.3192409

[62] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 821–839. doi:10.1145/2814270.2814275

[63] MIT Probabilistic Computing Project. 2025. *GenJAX: Source Code Repository*. https://github.com/probcomp/genjax

[64] Kevin P. Murphy and Mark A. Paskin. 2001. Linear-time inference in Hierarchical HMMs. In *Advances in Neural Information Processing Systems 14: NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada*, Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani (Eds.). MIT Press, Vancouver, BC, Canada, 833–840. https://proceedings.neurips.cc/paper/2001/hash/aebf7782a3d445f43cf30ee2c0d84dee-Abstract.html

[65] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 84)*. PMLR, Lanzarote, Canary Islands, 1037–1046. doi:10.48550/arXiv.1708.07787

[66] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer, Cham, Switzerland, 62–79. doi:10.1007/978-3-319-29604-3_5

[67] Radford M Neal. 2011. MCMC using Hamiltonian dynamics. In *Handbook of Markov chain Monte Carlo*. Chapman and Hall/CRC, Boca Raton, FL, USA, 113–162. doi:10.48550/arXiv.1206.1901

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). Curran Associates, Inc., Vancouver, BC, Canada, 8024–8035. doi:10.48550/arXiv.1912.01703

[69] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *CoRR* abs/1912.11554 (2019). doi:10.48550/arXiv.1912.11554 arXiv:1912.11554 [cs.LG]

[70] Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–13. doi:10.1109/InPar.2012.6339601

[71] Rajesh Ranganath, Sean Gerrish, and David M. Blei. 2014. Black Box Variational Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 33)*. PMLR, Reykjavik, Iceland, 814–822. doi:10.48550/arXiv.1401.0118

[72] Feras Saad, Brian Patton, Matthew Douglas Hoffman, Rif A. Saurous, and Vikash Mansinghka. 2023. Sequential Monte Carlo Learning for Time Series Structure Discovery. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, Honolulu, HI, USA, 29473–29489. doi:10.48550/arXiv.2307.09607

[73] Feras A. Saad, Marco Cusumano-Towner, and Vikash K. Mansinghka. 2022. Estimators of Entropy and Information via Inference in Probabilistic Models. *CoRR* abs/2202.12363 (April 2022). doi:10.48550/arXiv.2202.12363 arXiv:2202.12363 [cs.LG]

[74] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 37:1–37:32. doi:10.1145/3290350

[75] Feras A. Saad, Cameron E. Freer, Nathanael L. Ackerman, and Vikash K. Mansinghka. 2019. A Family of Exact Goodness-of-Fit Tests for High-Dimensional Discrete Distributions. *CoRR* abs/1902.10142 (Feb. 2019). doi:10.48550/

arXiv.1902.10142 arXiv:1902.10142 [cs.LG]

[76] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2063384.2063405

[77] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (April 2016), e55. doi:10.7717/peerj-cs.55 Publisher: PeerJ Inc..

[78] Adam Ščibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational validation of higher-order Bayesian inference. *Proc. ACM Program. Lang.* 2, POPL (2017), 60:1–60:29. doi:10.1145/3158148

[79] Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning proposals for probabilistic programs with inference combinators. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. PMLR, Virtual, 1056–1066. doi:10.48550/arXiv.2103.00668 ISSN: 2640-3498.

[80] Walid Taha. 1999. Multi-stage programming: its theory and applications. In *Applied Semantics Summer School*. Springer, Caminha, Portugal, 145–174.

[81] Luke Tierney. 1994. Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics* 22, 4 (Dec. 1994), 1701–1728. doi:10.1214/aos/1176325750 Publisher: Institute of Mathematical Statistics.

[82] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. *ACM Transactions on Programming Languages and Systems* 40, 4 (2016), 1–46. doi:10.1145/3064899

[83] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, Toulon, France. doi:10.48550/arXiv.1701.03757

[84] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *CoRR* abs/1610.09787 (2016). doi:10.48550/arXiv.1610.09787 arXiv:1610.09787 [stat.ML]

[85] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam Craig Pocock, Stephen J. Green, and Guy L. Steele Jr. 2014. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems 27 (NIPS 2014)*. Curran Associates, Inc., Montréal, Canada, 2600–2608. doi:10.48550/arXiv.1312.3613

[86] David Wingate, Andreas Stuhlmueller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, Fort Lauderdale, FL, USA, 770–778. https://proceedings.mlr.press/v15/wingate11a.html ISSN: 1938-7228.

[87] Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. 2023. From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought. *CoRR* abs/2306.12672 (2023). doi:10.48550/arXiv.2306.12672 arXiv:2306.12672 [cs.CL]

[88] Lance Ying, Tan Zhi-Xuan, Lionel Wong, Vikash Mansinghka, and Joshua Tenenbaum. 2024. Grounding Language about Belief in a Bayesian Theory-of-Mind. doi:10.48550/arXiv.2402.10416 arXiv:2402.10416 [cs].

[89] Tan Zhi-Xuan, Jordyn Mann, Tom Silver, Josh Tenenbaum, and Vikash Mansinghka. 2020. Online Bayesian Goal Inference for Boundedly Rational Planning Agents. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., Virtual, 19238–19250. doi:10.48550/arXiv.2006.07532

[90] Guangyao Zhou, Nishad Gothoskar, Lirui Wang, Joshua B. Tenenbaum, Dan Gutfreund, Miguel Lázaro-Gredilla, Dileep George, and Vikash K. Mansinghka. 2023. 3D Neural Embedding Likelihood: Probabilistic Inverse Graphics for Robust 6D Pose Estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Paris, France, 21559–21569. doi:10.1109/ICCV51070.2023.01977

Received 2025-07-10; accepted 2025-11-06