Gen: A General-Purpose Probabilistic Programming System with Programmable Inference

Marco F. Cusumano-Towner
Massachusetts Institute of Technology
USA

Feras A. Saad
Massachusetts Institute of Technology
USA

Alexander K. Lew
Massachusetts Institute of Technology
USA

Vikash K. Mansinghka
Massachusetts Institute of Technology
USA

Abstract

Although probabilistic programming is widely used for some restricted classes of statistical models, existing systems lack the flexibility and efficiency needed for practical use with more challenging models arising in fields like computer vision and robotics. This paper introduces Gen, a general-purpose probabilistic programming system that achieves modeling flexibility and inference efficiency via several novel language constructs: (i) the generative function interface for encapsulating probabilistic models; (ii) interoperable modeling languages that strike different flexibility/efficiency tradeoffs; (iii) combinators that exploit common patterns of conditional independence; and (iv) an inference library that empowers users to implement efficient inference algorithms at a high level of abstraction. We show that Gen outperforms state-of-the-art probabilistic programming systems, sometimes by multiple orders of magnitude, on diverse problems including object tracking, estimating 3D body pose from a depth image, and inferring the structure of a time series.

CCS Concepts • **Mathematics of computing → Probabilistic reasoning algorithms**.

Keywords Probabilistic programming, Markov chain Monte Carlo, sequential Monte Carlo, variational inference

ACM Reference Format:
Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314642

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314642

1 Introduction

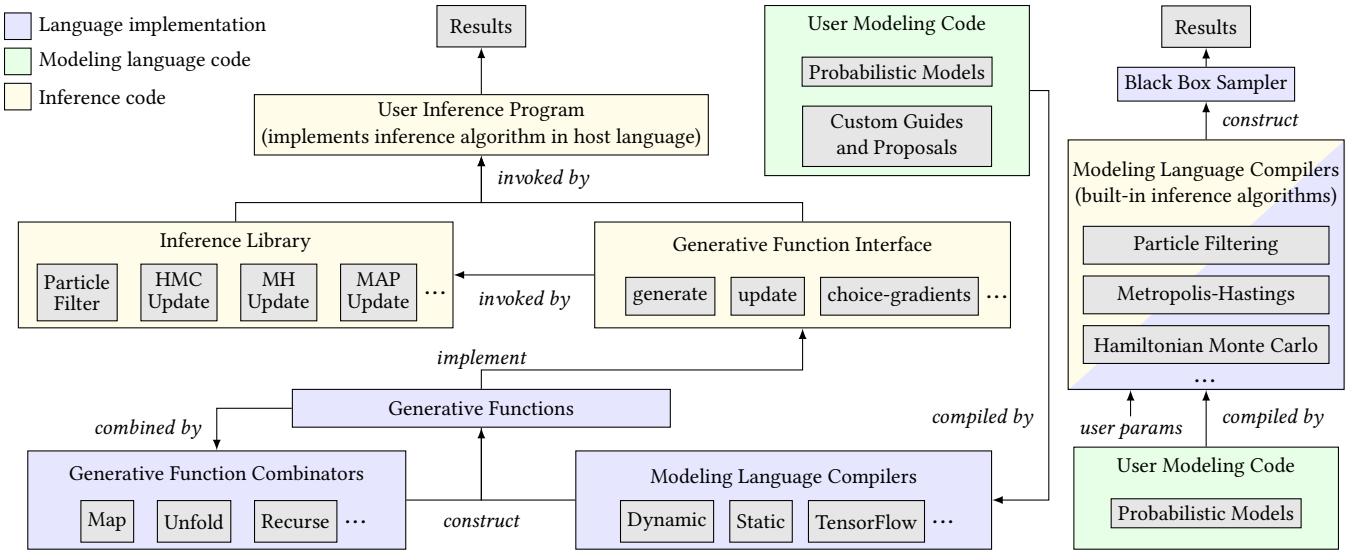
Probabilistic modeling and inference are core tools in diverse fields including statistics, machine learning, computer vision, cognitive science, robotics, natural language processing, and artificial intelligence [9, 20, 27, 36, 44, 53, 60]. In the probabilistic modeling and inference paradigm, models correspond to probability distributions, and inference algorithms (often approximately) perform standard operations on distributions, such as conditioning and optimization. To meet the functional requirements of applications, practitioners use a broad range of modeling techniques and approximate inference algorithms. However, implementing inference algorithms is often difficult and error prone. Probabilistic programming systems formalize and simplify the use of probabilistic modeling and inference, by providing *modeling languages* in which users express models and high-level programming constructs that automate aspects of inference.

However, existing systems are impractical for general-purpose use. Some systems provide restricted modeling languages that are only suitable for specific problem domains. Others provide 'universal' modeling languages that can represent any model, but support only a limited set of inference algorithms that converge prohibitively slowly. This paper addresses the problem of building a practical, general-purpose probabilistic programming system that achieves both *modeling expressiveness* and *inference efficiency*, producing fast and accurate inference results in diverse problem domains.

1.1 Key Challenges

This section explores the challenges that expressive modeling languages pose for inference efficiency. We distinguish between two kinds of inference efficiency: *inference algorithm efficiency* (how many iterations or particles an inference algorithm needs to produce accurate results), and *implementation efficiency* (how long it takes to run inference for a single iteration or with a single particle). We discuss the unique challenges associated with each.

Inference Algorithm Efficiency Efficient inference typically requires tailoring and tuning the algorithm to the particular model or inference problem. Languages that support a limited model class and only a few inference algorithms



(a) Gen's Architecture **(b)** Standard Architecture

Figure 1. Comparison of Gen's architecture to a standard probabilistic programming architecture.

can perform some of this tuning automatically to improve efficiency [8, 62]. In contrast, tailoring an algorithm for a model written in a more expressive language is an open-ended and difficult-to-automate task, which may require combining multiple custom inference strategies [40]. The approach taken in Gen to achieve inference algorithm efficiency is to provide high-level abstractions that enable users to specify custom inference algorithms themselves, while automating optimization, parameter-tuning, and the calculation of probability ratios used in Monte Carlo and variational inference. A key challenge is designing these abstractions so that they are flexible, fast, and easy to use.

Implementation Efficiency A given inference algorithm may converge in a small number of steps, but each step may take a long time to run if the algorithm is implemented using an inefficient system. Implementation efficiency is determined by factors such as the data structures used to store algorithm state, and whether the system exploits caching and incremental computation. Just as dynamic language features can make it difficult to produce optimized machine code in a traditional programming language setting, flexible modeling languages can make it difficult to produce efficient implementations of inference algorithms, which typically leverage static analysis of control flow and dependency relationships between model variables. Therefore, a second key challenge is finding representations for models that allow the system to exploit their special structure whenever possible. The traditional choice, to represent models as program code in an expressive or even Turing-complete modeling language, makes such optimizations difficult, due to the general intractability of detecting non-trivial program properties statically in arbitrary program code.

1.2 This Work

To address these challenges, we introduce a novel architecture (Figure 1) for probabilistic programming systems. Our core insight is to represent models not as program code in a Turing-complete modeling language (Figure 1b), but as black boxes that expose capabilities useful for inference via a common interface (Figure 1a). We call these black box models *generative functions* and we call the interface they satisfy the *generative function interface* (GFI). Leveraging this interface, we provide tools for (i) constructing models that support efficient implementations of inference operations, and (ii) tailoring inference to improve inference algorithm efficiency.

Tools for Constructing Models We provide multiple interoperable modeling languages, each striking a different flexibility/efficiency trade-off. A single model can combine code from multiple modeling languages. Compilers for these languages generate implementations of the GFI methods that use model-specific logic for performing common inference operations more efficiently. The resulting generative functions leverage data structures well-suited to the model as well as incremental computation. We also provide model combinators, which capture common modeling patterns involving repeated computation (e.g., a dataset of independently sampled data points) and construct generative functions with inference code that exploits conditional independence.

Tools for Tailoring Inference We provide a high-level library for inference programming, which implements inference algorithm building blocks that interact with models

only through the GFI. The library supports a degree of flexibility that is unmatched by existing systems, for example enabling users to design and train custom multi-site Markov chain Monte Carlo proposals with automatic (and efficient) calculation of acceptance probabilities.

Evaluation We evaluate our system, Gen, on several challenging inference problems: estimating 3D body pose from a depth image; robust regression; inferring, in a simulator-based model, the probable destination of a person or robot traversing its environment; and structure learning for time series data. In each case, Gen outperforms existing probabilistic programming systems, often by one or more orders of magnitude. These gains are enabled by a combination of Gen's (i) inference algorithm efficiency, due to its flexible inference programming capabilities, and (ii) implementation efficiency, due to its generative function combinators and efficient modeling language implementations.

Main Contributions The contributions of this paper are:

- Gen, a probabilistic programming system embedded in Julia [4], in which users (i) define models in embedded modeling languages and (ii) implement high-level *inference programs* in the host language (Section 3).
- The *generative function interface*, a black box abstraction for probabilistic models that separates logic for efficient computation from inference algorithm design (Section 4).
- Three interoperable modeling languages: one that is Turing-complete for maximal model expressiveness, one that is amenable to static analysis, and one based on TensorFlow.
- Three *generative function combinators* (*map*, *unfold*, *recurse*), which produce generative functions that exploit common patterns of conditional independence (Section 5).
- An empirical evaluation of Gen's modeling expressiveness and inference efficiency relative to other systems on five challenging inference problems (Section 7).

2 Background

This section briefly introduces probabilistic modeling and inference and describes classes of inference tasks that exemplify some of the target use cases for Gen.

2.1 Probabilistic Modeling and Inference

Probabilistic modeling and inference is central to various fields including computer vision [27], robotics [60], and natural language processing [36].

Generative Models In probabilistic modeling and inference, uncertain knowledge about a domain is often represented by a joint probability distribution on variables whose values are typically known (*observed variables*) and variables whose values are typically unknown (*latent variables*). Such probability distributions, known as *generative models*[1], are typically constructed from a combination of prior knowledge and empirical evidence.

Inference Tasks Probability theory provides specifications for various tasks involving generative models including inferring the values of latent variables from values of observed variables given an assumed generative model. This task, which is called *posterior inference* in the context of Bayesian statistics, requires conditioning the joint distribution over latent and observed variables on the event that the observed variables took certain values. The latent variables may include numerical variables (often called *parameters*) as well as combinatorial discrete objects (often called *structure*). Inferences about the latent variables may take the form of (i) samples from the conditional probability distribution on the latent variables, (ii) an analytic representation of the conditional distribution, or (iii) some summary of the conditional distribution, such as the single value with maximum probability (as in *maximum a posteriori* or 'MAP' inference).

2.2 Algorithms for Probabilistic Inference

The probabilistic approach to reasoning defines the mathematical specification for solutions to inference tasks, but efficient algorithms for computing these solutions are only known for models that possess special conditional independence or analytic structure (e.g. discrete-variable tree-structured graphical models, linear-Gaussian state-space models, and log-concave likelihood functions). Therefore, a large variety of approximation algorithms have been devised. In this paper we focus on two classes of approximation algorithms for inference with generative models that are broadly applicable and are active areas of research: Monte Carlo and variational inference algorithms.

Monte Carlo Inference Monte Carlo inference algorithms approximate a conditional probability distribution using samples from the distribution or an approximation to the distribution. Monte Carlo algorithms include rejection sampling, Markov chain Monte Carlo (MCMC), importance sampling, and sequential Monte Carlo (SMC) including particle filtering [15]. Monte Carlo approaches provide asymptotically exact results and place few requirements on the structure of models. The efficiency of a Monte Carlo algorithm is determined by the choice of *proposal distribution*, which encodes knowledge about the inference problem that is provided by the user or learned automatically through adaptation.

Variational Inference Variational inference algorithms approximate a conditional probability distribution with a probability mass or density function that is optimized using numerical methods to match as closely as possible with the desired distribution [64]. Typically, variational techniques do not provide asymptotically exact results, although variational techniques are often considered more scalable than Monte Carlo methods. Recently, a number of techniques that

[1] Gen also supports discriminative models and supervised learning, but these are not the focus of this paper.

combine use of Monte Carlo and variational approaches have been introduced (e.g. [55]).

2.3 Examples

The remainder of this section introduces five challenging classes of inference tasks and approximate inference approaches for each task. These inference tasks form the basis for the evaluation of Gen (Section 7).

Hierarchical Bayesian Statistics Hierarchical Bayesian statistical models specify a joint probability distribution over latent variables at two or more levels of abstraction. For example, individual data points may be associated with 'local' latent variables, whose prior distributions are in turn parameterized by shared 'global' latent variables that have prior distributions of their own [6]. Examples include Bayesian mixture models and topic models like latent Dirichlet allocation [7]. MCMC and Gibbs sampling [21] in particular have been used for inference in hierarchical Bayesian models for decades [19]. Variational inference, often based on the mean-field assumption, is also popular, particularly for 'big data' applications with millions of data points [6].

MCMC for Bayesian Structure Learning One approach to inferring the structure of a model from data involves placing a prior distribution on the set of possible structures and jointly inferring the structure and the numerical parameters using MCMC for posterior inference [2, 38, 67]. Distributions from Bayesian nonparametrics are often used to construct prior distributions on infinite sets of possible model structures, such as mixture models with an unbounded number of clusters [47], or directed graphs with an unknown number of nodes [67]. Recent work represents model structures as elements of formal languages and interprets Bayesian structure learning as a form of program synthesis [54].

Monte Carlo Techniques for Simulation-Based Models Many Bayesian statistical models used in the natural sciences are based on stochastic simulators that are not amenable to analytic inference approaches [65]. The task is to infer the parameters that were fed into the simulator, given a prior distribution on the parameters and an observed data set that is assumed to be the simulator output. Inference in such models, often based on Monte Carlo, has running times that are dominated by runs of the simulator. Stochastic simulators have also been used to model the behavior of autonomous agents for motion forecasting and goal inference [12, 28].

Particle Filtering for State Estimation Tracking the dynamic state (e.g. location) of an object over time from noisy sensor measurements using probabilistic models of the object dynamics and measurement noise is a well-studied problem with applications in various fields including autonomous robotics [60]. Exact inference is usually only possible for restricted classes of dynamics and measurement models, such as the linear-Gaussian models supported by the Kalman filter [32]. Particle filtering is an online sequential Monte Carlo algorithm that supports essentially arbitrary dynamics and measurement models, and can represent posterior distributions with multiple modes, unlike the Kalman filter and its most popular extensions [31]. A key design parameter in a particle filter is the proposal distribution that generates new states at each time step. A generic choice of proposal is forward sampling from the dynamics model, but custom proposals that take into account the latest available measurement can provide orders-of-magnitude improvement in efficiency (e.g. compare FastSLAM and FastSLAM 2.0 [43]).

Computer Vision via Inverse Graphics Given an image of an object of a known class, an important task in computer vision is to infer the 3D pose parameters of the object relative to the camera, as well as any intrinsic parameters of the object (e.g. the dimensions a table). Popular object classes for applications include furniture and articulated human bodies and hands [13, 58]. A modern 'inverse graphics' approach to this task uses generative probabilistic models that (i) sample object parameters from a prior distribution, (ii) produce a 3D mesh from the object's intrinsic parameters, and (iii) render an image of the object using a graphics engine. Then, object parameters are estimated from an observed image using posterior inference in the generative model [14, 39]. An emerging approach to inference in these models uses 'data-driven' custom proposal distributions trained using supervised learning on joint samples from the generative model to accelerate Monte Carlo algorithms [30, 35].

3 Overview

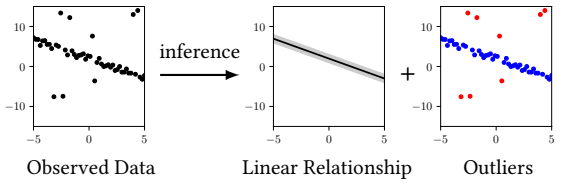
In this section, we illustrate Gen's programming style by walking through an example generative model and two user inference programs for the model. The example in Figure 2 is based on modeling a response variable y as a linear function of an independent variable x. Figure 2a presents the inference problem: given a data set of observed pairs (x_i, y_i), the task is to infer the relationship between y and x, as well as identify any data points i that do not conform to the inferred linear relationship (i.e. detect outliers). Figure 2b shows the generative model. Figures 2f and 2g show two user inference programs implemented in Julia. Each of these inference programs implements a different algorithm for solving the inference task and, as a result, exhibits different efficiency characteristics, shown in Figure 2h.

3.1 Example Generative Model

We now describe the ideas in Figure 2b in greater detail.

Generative Models are Generative Functions Users define generative models by constructing *generative functions*. Here, we build a generative function that represents a joint probability distribution over several random variables: the

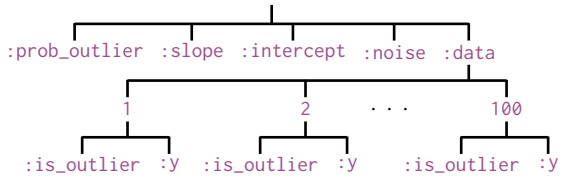
(a) Probabilistic inference task



(b) Generative model

```
@gen function generate_datum(x, prob_outlier, noise, (grad)(slope),
        (grad)(intercept))::Float64
  if @trace(bernoulli(prob_outlier), :is_outlier)
    (mu, std) = (0., 10.)
  else
    (mu, std) = (x * slope + intercept, noise)
  end
  return @trace(normal(mu, std), :y)
end

generate_data = Map(generate_datum)

@gen (static) function model(xs)
  slope = @trace(normal(0, 2) :slope)
  intercept = @trace(normal(0, 2) :intercept)
  noise = @trace(gamma(1, 1) :noise)
  prob_outlier = @trace(uniform(0, 1) :prob_outlier)
  n = length(xs)
  ys = @trace(generate_data(xs, fill(prob_outlier, n),
      fill(noise, n), fill(slope, n), fill(intercept,n)), :data)
  return ys
end
```

(c) The hierarchical address space of `model` in (b).



(d) Helper Julia function for making observations

```
function make_observations(ys::Vector{Float64})
  observations = choicemap()
  for i=1:length(ys)
    observations[:data => i => :y] = ys[i]
  end
  return observations
end
```

(e) Custom proposal used by inference programs (f) and (g)

```
@gen function is_outlier_proposal(previous_trace, i::Int)
  is_outlier = previous_trace[:data => i => :is_outlier]
  p_outlier = is_outlier ? 0.0 : 1.0
  @trace(bernoulli(p_outlier), :data => i => :is_outlier)
end
```

(f) User inference program 1

```
function inference_program_1(xs::Vector{Float64}, ys::Vector{Float64})
  # constrain observed random variables to their observed values
  constraints = make_observations(ys)

  # obtain an execution trace of model that satisfies the constraints
  (trace, _) = generate(model, (xs,), constraints)

  # iteratively update the values of latent variables
  for iter=1:100
    # update four parameters all at once
    selection = select(:prob_outlier, :noise, :slope, :intercept)
    (trace, _) = metropolis_hastings(trace, selection)

    # update outlier classifications one by one, using a custom proposal
    for i=1:length(xs)
      (trace, _) = metropolis_hastings(trace, is_outlier_proposal, (i,))
    end
  end

  return trace
end
```

(g) User inference program 2

```
function inference_program_2(xs::Vector{Float64}, ys::Vector{Float64})
  # constrain observed random variables to their observed values
  constraints = make_observations(ys)

  # initialize slope and intercept using least squares regression
  (slope, intercept) = least_squares(xs, ys)
  constraints[:slope] = slope
  constraints[:intercept] = intercept

  # obtain an execution trace of model that satisfies the constraints
  (trace, _) = generate(model, (xs,), constraints)

  # iteratively update the values of latent variables
  for iter=1:100
    # update custom subsets of parameters, repeating several times
    for j=1:5
      (trace, _) = metropolis_hastings(trace, select(:prob_outlier))
      (trace, _) = metropolis_hastings(trace, select(:noise))
      (trace, _) = mala(trace, select(:slope, :intercept), 0.001)
    end

    # update outlier classifications one by one, using a custom proposal
    for i=1:length(xs)
      (trace, _) = metropolis_hastings(trace, is_outlier_proposal, (i,))
    end
  end

  return trace
end
```

(h) Runtime versus accuracy of inference programs in (f) and (g).

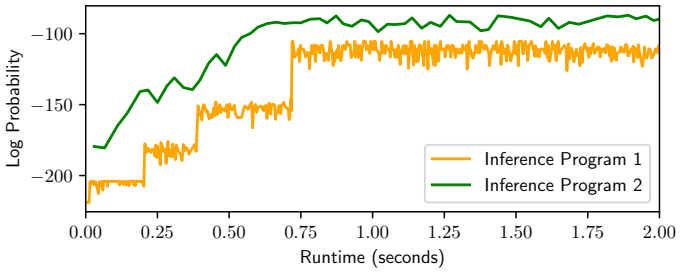


Figure 2. Example of modeling and inference programming in Gen, for a regression task (a). Users define generative models (b) and custom proposal distributions (e) in embedded modeling languages. Users also implement inference programs (f)–(g) in the host language (Julia), using methods provided by the inference library (shown in bold). (h) shows the time-accuracy profiles of the two inference programs, which implement inference algorithms that have different efficiency characteristics.

line parameters `slope` and `intercept`, the noise parameters `prob_outlier` and `noise`, and the vectors of per-data-point random variables $\texttt{is\_outlier}_{1\ldots n}$ and $\texttt{y}_{1\ldots n}$. We do so in three steps, using three of Gen's interoperable tools for constructing models: two modeling languages, and the generative function combinator `Map`.

Dynamic Modeling Language First, we use Gen's Turing-complete *dynamic modeling language* to write the generative function `generate_datum` (lines 1-9), which represents the conditional distribution of `is_outlier` and `y` given an `x` coordinate, the line parameters, and the noise parameters. Generative functions in this modeling language are identified by an `@gen` annotation in front of a regular Julia function definition, and describe a distribution implicitly by sampling from it. The `generate_datum` function samples the random variable `is_outlier` from a `bernoulli` distribution with parameter `prob_outlier`. Then, depending on the result, it samples `y` either near the line defined by `slope` and `intercept`, or from a wide "outlier" distribution centered at 0.

Generative Function Combinators We next apply the `Map` combinator (line 11), which accepts as input a generative function (`generate_datum`) and produces a second generative function (`generate_data`), which applies the first function repeatedly to a vector of argument values, using independent randomness for each application. The `generate_data` function represents the conditional distribution of the random *vectors* $\texttt{is\_outlier}_{1\ldots n}$ and $\texttt{y}_{1\ldots n}$, given line parameters, noise parameters, and a vector `xs` of input values.

Static Modeling Language We are finally ready to express our complete model as a generative function, `model`, representing a distribution over *all* of our random variables, parameterized by a vector of inputs `xs` (lines 13-22). We do so using the *static modeling language* (by using the annotation `(static)` on line 13), which prohibits control flow constructs like `if` and `for`, but generates efficient inference code that leverages specialized data structures and incremental computation, and which can still call other generative functions (like `generate_data`) that were not written in the static modeling language. The `model` function first samples parameter values for `slope`, `intercept`, `noise`, and `prob_outlier`, then calls `generate_data` to sample $\texttt{is\_outlier}_{1\ldots n}$ and $\texttt{y}_{1\ldots n}$ conditioned on these parameters.

Addresses of Random Variables Each random variable (or 'random choice') that a generative function samples has a unique address (shown in purple). Within the static and dynamic modeling languages, addresses are assigned with the `@trace` keyword. For example, the `prob_outlier` choice (line 17) has address `:prob_outlier`. In general, the address of a random choice is independent of the name to which it is assigned in the function body (random choices need not even be bound to any identifier in the function body). In the dynamic modeling language, addresses are Julia values that are dynamically computed during function execution, whereas in the static language, they must be literal symbols. Generative functions produced by combinators have their own internal addressing schemes, as described below.

Hierarchical Address Spaces Addresses can be simple Julia values like symbols or integers, or compound, hierarchical *paths*, like `:a => :b => :c`. These hierarchical addresses can arise in two ways. First, when calling other generative functions, code in the dynamic or static modeling languages can use `@trace` to indicate the *namespace* relative to which all random choices made during the call are addressed. For example, `model` calls `generate_data` under the namespace `:data` (lines 19-20). Second, generative functions produced by combinators also give a namespace to each call they make—`generate_data` calls `generate_datum` once for each data point with a unique integer namespace ranging from 1 to 100 (for 100 data points). These namespaced calls result in hierarchical address spaces, as shown in Figure 2c. For example, `:data => 2 => :is_outlier` refers to the `is_outlier` random choice for the second data point.

3.2 Example User Inference Programs

Figures 2f and 2g show two user inference programs, which are Julia functions that implement different custom algorithms for inference in the generative model. The inference programs take as input a data set containing many (x_i, y_i) values and return inferred values for the parameters that govern the linear relationship and the outlier classifications. The inference programs invoke high-level functions provided by Gen, which are shown in bold-face (e.g **`select`**, **`generate`**).

Execution Traces The results of inference take the form of an *execution trace* (`trace`) of the generative function `model`. An execution trace of a generative function contains the values of random choices made during an execution of the function, and these values are accessed using indexing syntax (e.g. `trace[:slope]`, `trace[:data => 5 => :is_outlier]`). The inference programs return traces whose random choices represent the inferred values of latent variables in the model. In addition to the random choices, an execution trace (or just 'trace') contains the arguments on which the function was executed and the function's return value. Traces appear immutable to the user.

Generative Function Interface The inference program in Figure 2f obtains an initial trace by calling the **`generate`** method on the generative function (`model`) and supplying: (i) arguments (`xs`) to the generative function; and (ii) a mapping (`constraints`) from addresses of certain random choices to constrained values. The **`generate`** method is part of the *generative function interface* (GFI, Section 4), which exposes low-level operations on execution traces. The `constraints` are a *choice map* (produced by the Julia function `make_observations` in Figure 2d), which is a prefix tree that maps addresses to

values. The **generate** method produces traces whose random choices satisfy given constraints. Both inference programs constrain the y-coordinates to their observed values. In inference program 1 (Figure 2f), the **generate** method samples values for all the latent variables (the unconstrained choices) from the prior distribution. Inference program 2 (Figure 2g) instead initializes the `slope` and `intercept` (lines 62-63) to values obtained from the Julia function `least_squares` (implementation not shown).

Inference Library After obtaining an initial trace, the user inference programs in Figures 2f and 2g then repeatedly apply a cycle of different Markov Chain Monte Carlo (MCMC) updates to various random choices (lines 42–52, 68–81). These updates change the values of the latent variables so that they better explain the observations. The MCMC updates use the functions **metropolis_hastings** and **mala** from Gen's built-in *inference library*, which provides higher-level building blocks for inference algorithms that are built on top of the GFI (e.g. `mala` implements a Metropolis-Adjusted Langevin Algorithm [52] update). Inference program 1 uses a variant of **metropolis_hastings** that proposes new values for certain selected random choices according to a generic proposal distribution and accepts or rejects the proposed change according to the Metropolis-Hastings (MH, [26]) rule.

Custom Proposals are Generative Functions The inference programs in Figures 2f and 2g update the outlier indicator variables by calling a variant of **metropolis_hastings** (lines 50, 79) from the inference library that accepts a custom proposal distribution represented by a generative function. We perform an MH update to each `:is_outlier` choice using the custom proposal distribution `is_outlier_proposal` in Figure 2e, which reads the previous value of the variable and then proposes its negation by making a random choice at the same address `:data => i => :is_outlier`. Custom proposal distributions are generative functions written in the same modeling languages used for generative models.

Using Host Language Abstraction Inference programs use familiar host language constructs, like procedural abstraction (such as reusable Julia code like `make_observations` and user-defined Gen proposals like `is_outlier_proposal`) and loop constructs (e.g. to repeatedly invoke nested MCMC and optimization updates). These host language features obviate the need for new and potentially more restrictive 'inference DSL' constructs [18, 40]. Using the host language for inference programming also allows for seamless integration of external code into inference algorithms, illustrated by the least-squares initialization in inference program 2 (line 61).

Performance Depends on the Inference Algorithm As seen in Figure 2h, the two inference programs have different time-accuracy profiles. The generic proposal for the parameters used in inference program 1 causes slow convergence compared to inference program 2. The constructs in Gen make it practical for the user to experiment with and compare various inference strategies for a given model.

4 Generative Function Interface

The *generative function interface* (GFI) is a black box abstraction for probabilistic models that provides a barrier between the implementation of modeling languages and the specification of inference algorithms. *Generative functions* are objects that implement the methods in the GFI and are typically produced by a modeling language compiler. For example, the dynamic modeling language compiler takes a `@gen` function definition and produces a generative function that implements the GFI. The GFI allows the user to express a wide variety of inference algorithms and algorithm customizations, while enabling model-specific optimizations for implementation efficiency, such as incremental computation.

4.1 Formalizing Generative Functions

Choice Map A *choice map* s is a map from a set of addresses A to a set of values V, where $s(a)$ denotes the value assigned to address $a \in \mathsf{A}$. The special value $s(a) = \bot$ indicates that s does not assign a meaningful value to a. We define $\mathrm{dom}[s] \coloneqq \{a \in \mathsf{A} \mid s(a) \neq \bot\}$ of s as the set of non-vacuous addresses in s. Let $s|_B$ be the choice map obtained by restricting s to a subset of addresses $B \subseteq \mathsf{A}$, i.e. $s|_B(a) = s(a)$ if $a \in B$ and $s|_B(a) = \bot$ otherwise. For two choice maps s and t, let $s \cong t$ mean $s(a) = t(a)$ for all $a \in \mathrm{dom}[s] \cap \mathrm{dom}[t]$. Finally, we let $\mathsf{S} \coloneqq \mathsf{V}^{\mathsf{A}}$ be the set of all choice maps from A to V.

Generative Function A generative function $\mathcal{G}$ is a tuple $\mathcal{G} = (\mathsf{X}, \mathsf{Y}, f, p, q)$. Here, X and Y are sets that denote the domain of the arguments and the domain of the return value of the function, respectively. Moreover, $p : \mathsf{X} \times \mathsf{S} \to [0, 1]$ is a family of probability distributions on choice maps $s \in \mathsf{S}$ indexed by argument $x \in \mathsf{X}$. In words, x are arguments to the generative function and s contains values for all the random choices made during an execution on those arguments. Since p is a distribution, for each x, we have $\sum_{s \in \mathsf{S}} p(x, s) = 1$. We use the notation $p(s; x) \coloneqq p(x, s)$ to separate the arguments x from the choice map s. The output function $f : \{(x, s) : x \in \mathsf{X}, p(s; x) > 0\} \to \mathsf{Y}$ maps an argument x and a choice map s (containing all of the sampled random choices) to a return value $f(x, s) \in \mathsf{Y}$. Finally, q is a family of *internal proposal distributions* that assigns a probability $q(s; x, u)$ to each choice map $s \in \mathsf{S}$ for all (x, u) where $x \in \mathsf{X}$ and where $\exists t \in \mathsf{S}$ such that $p(t; x) > 0$ and $t \cong u$. The proposal distribution q must satisfy (i) $\sum_{s \in \mathsf{S}} q(s; x, u) = 1$ for all such (x, u); and (ii) $q(s; x, u) > 0$ if and only if $s \cong u$ and $p(s; x) > 0$.

4.2 Interface Methods

We now describe several GFI methods. Each method takes a generative function $\mathcal{G}$ or a trace, as well as method-specific arguments. We will denote a choice map by t if $p(t; x) > 0$ (for some $x \in \mathsf{X}$) when the choice map is 'complete' and by

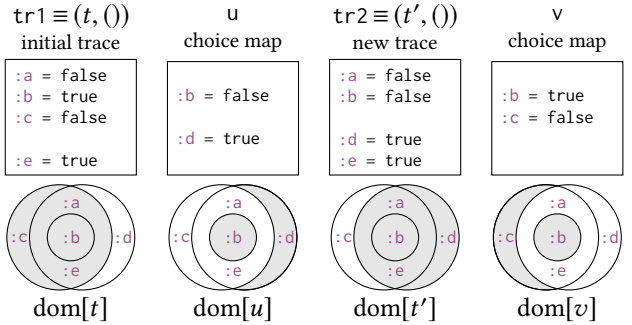
u when the choice map is 'partial'. Because traces contain a complete choice map and arguments, they are denoted (t, x).

`generate` *(obtaining a trace subject to constraints)* This method takes a choice map u, arguments x, and returns: (i) a trace (t, x) such that $t \cong u$, sampled using the internal proposal distribution (denoted $t \sim q(\cdot; x, u)$); as well as (ii) a weight $w := p(t; x)/q(t; x, u)$.

`propose` *(proposing a choice map)* This method samples a choice map from a generative function $\mathcal{G}$ that is used as a proposal distribution and computes the proposal probability. In particular, `propose` takes arguments x, and returns a choice map $t \sim p(\cdot; x)$ and a weight $w = p(t; x)$.

`assess` *(assessing the probability of a choice map)* This method computes (or estimates) the probability that a generative function $\mathcal{G}$ produces a given choice map u. It is equivalent to `generate` except that it only returns the weight w, and not the trace. Note that w is either the exact probability that the choice map u is produced by $\mathcal{G}$ (on arguments x) or an unbiased estimate of this probability.

```
@gen function foo()
    val = @trace(bernoulli(0.3), :a)
    if @trace(bernoulli(0.4), :b)
        val = @trace(bernoulli(0.6), :c) && val
    else
        val = @trace(bernoulli(0.1), :d) && val
    end
    val = @trace(bernoulli(0.7), :e) && val
    return val
end

# Sample a random initial trace (tr1).
x = ()                          # empty tuple, foo takes no args
u = choicemap()                 # empty choice map
(tr1, w1) = generate(foo, x, u) # 'tr1' is the trace

# Use 'update' to obtain a trace (tr2) with :b=false, :d=true.
argdiff = () # there are no arguments, so argdiff is the empty tuple
u =  choicemap((:b, false), (:d, true))
(tr2, w2, retdiff, v) = update(tr1, x, argdiff, u)
```



$$p(t; x) = 0.7 \times 0.4 \times 0.4 \times 0.7 = 0.0784$$
$$p(t'; x') = 0.7 \times 0.6 \times 0.1 \times 0.7 = 0.0294$$
$$w = p(t'; x')/p(t; x) = 0.0294/0.0784 = 0.375$$

Figure 3. Illustration of `update` on a trace of a simple generative function `foo`. Shaded regions indicate sets of addresses.

`update` *(updating a trace)* The `update` method makes adjustments to random choices in an execution trace (t, x) of $\mathcal{G}$, which is a common pattern in iterative inference algorithms such MCMC and MAP optimization. In particular, `update` takes a trace (t, x) of $\mathcal{G}$; new arguments x'; and a partial choice map u. It returns a new trace (t', x') such that $t' \cong u$ and $t'(a) = t(a)$ for $a \in (\text{dom}[t] \cap \text{dom}[t']) \setminus \text{dom}[u]$ and $\text{dom}[t'] \subseteq \text{dom}[t] \cup \text{dom}[u]$. That is, values are copied from t for addresses that remain and that are not overwritten by u. The method also returns a 'discard' choice map v where $v(a) := t(a)$ for $a \in (\text{dom}[t] \setminus \text{dom}[t']) \cup (\text{dom}[t] \cap \text{dom}[u])$. In words, v contains the values from the previous trace for addresses that were either overwritten by values in u or removed altogether. The method additionally returns a weight $w := p(t'; x')/p(t; x)$. Figure 3 illustrates the behavior of `update` for a trace of a toy generative function.

Often, in an invocation of `update`, the choice map u will contain values for only a small portion of the addresses in t and there will either be no change from x to x' or the change will be 'small' in some sense. In these cases, information about the change from x to x' can be used to make the implementation of `update` more efficient. To exploit this invocation pattern for performance gains, `update` accepts an *argdiff* value $\Delta(x, x')$ which contains information about the change in arguments (when available). To enable compositional implementations of `update`, it also returns a *retdiff* value $\Delta(y, y')$ that contains information about the change in return value (when available).

`choice_gradients` *(differentiation)* This method computes gradients of the log probability of a trace with respect to the values of the arguments x and/or the values of random choices. In particular, `choice_gradients` takes a trace (t, x) and a set of addresses B whose gradients are desired. It returns the gradient $\nabla_B \log p(t; x)$ of $\log p(t; x)$ with respect to the selected addresses as well as the gradient $\nabla_x \log p(t; x)$ with respect to the arguments. `choice_gradients` also accepts an optional return value gradient $\nabla_y J$ (for some function J) in which case it returns $\nabla_B(\log p(t; x)+J)$ and $\nabla_x(\log p(t; x)+J)$. For $\nabla_B \log p(t; x)$ to be defined, random choices in B must have differentiable density functions with respect to a base measure, and $p(t; x)$ is a joint density with respect to the induced base measure on choice maps, derived from the base measures for individual addresses. A formal measure-theoretic treatment of generative functions is future work.

4.3 Implementing a Metropolis-Hastings Update

The GFI provides building blocks for implementing probabilistic inference algorithms. For example, the inference library's implementation of a Metropolis-Hastings (MH) update with a custom proposal (used in Figure 2) is shown below. We call `propose` on the generative function `proposal` to generate the proposed choice map (`u`) and the forward proposal probability (`fwd_score`). The `update` method takes

the previous model trace (`trace`) and the proposed choice map, and returns (i) a new trace that is compatible with the proposed choice map; (ii) the model's contribution (`w`) to the MH acceptance probability; and (iii) the discard choice map (`v`), which would have to be proposed to reverse the move. Then, `assess` computes the reverse proposal probability (`rev_score`). Finally, we randomly accept or reject the change according to the MH acceptance probability (`alpha`). Note that all probabilities and weights are in log-space.

```
function metropolis_hastings(trace, proposal, proposal_args)
  proposal_args_fwd = (trace, proposal_args...,)
  (u, fwd_score) = propose(proposal, proposal_args_fwd)
  model_args = get_args(trace) # model arguments
  argdiff = map((_) -> NoChange(), model_args) # diffs for arguments
  (new_trace, w, _, v) = update(trace, model_args, argdiff, u)
  proposal_args_rev = (new_trace, proposal_args...,)
  rev_score = assess(proposal, proposal_args_rev, v)
  alpha = w + rev_score - fwd_score
  # accept (return new trace) or reject (return previous trace)
  r = uniform(0, 1)
  return log(r) < alpha ? (new_trace, true) : (trace, false)
end
```

4.4 Compositional Implementation Strategy

For the current built-in modeling languages, GFI methods are implemented by recursively calling the same GFI method for each generative function call site. Recall that the address hierarchy mirrors the call hierarchy (see e.g. Figure 2c). Execution traces are also hierarchical and have a 'sub-trace' for each generative function call in the call hierarchy. When a modeling language compiler produces a generative function $\mathcal{G}$, it can treat any generative function that is called by $\mathcal{G}$ as a black box that implements the GFI. This design enables (i) generative functions defined using different modeling languages to invoke one another and (ii) specialized GFI implementations for generative function combinators.

5 Generative Function Combinators

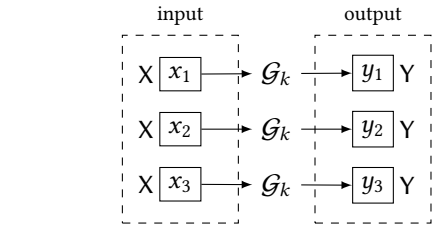
Iterative inference algorithms like MCMC or MAP optimization often make small adjustments to the execution trace of a generative function. For example, consider the call to `metropolis_hastings` in Figure 2f, Line 50. This call proposes a change to the `:is_outlier` choice for a single data point (`i`) and then invokes `update`. Because data points are conditionally independent given the parameters, an efficient implementation that exploits incremental computation would scale as $O(1)$, whereas a naive implementation that always visits the entire trace would scale as $O(n)$ where n is the number of data points.

To provide proper asymptotic scaling, we introduce *generative function combinators*, which provide efficient implementations of `update` (and other GFI methods) by exploiting common patterns of repeated computation and conditional independence that arise in probabilistic models. A generative function combinator takes as input a generative function $\mathcal{G}_k$, (called a "kernel") and returns a new generative function $\mathcal{G}'$ that repeatedly applies the kernel according to a particular pattern of computation. The GFI implementation for $\mathcal{G}'$ automatically exploits the static pattern of conditional independence in the resulting computation, leveraging *argdiff* and *retdiff* to provide significant gains in scalability. We next describe three examples of generative function combinators.

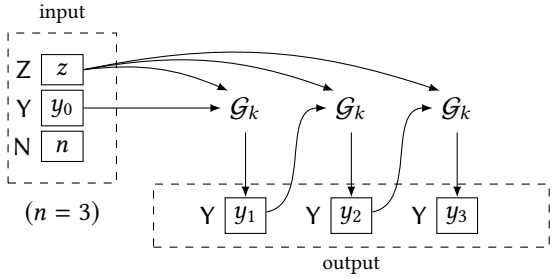
Map The `Map` combinator (Figure 4a) constructs a generative function $\mathcal{G}'$ that independently applies a kernel $\mathcal{G}_k$ from X to Y to each element of a vector of inputs $(x_1, \ldots, x_n)$ and returns a vector of outputs $(y_1, \ldots, y_n)$. Each application of $\mathcal{G}_k$ is assigned an address namespace $i \in \{1 \ldots n\}$. The implementation of `update` for `Map` only makes recursive GFI calls for the kernel on those applications i for which the choice map u contains addresses under namespace i, or for which $x_i \neq x_i'$. The `update` method accepts an *argdiff* value $\Delta(x, x')$ that indicates which applications i have $x_i \neq x_i'$, and a nested *argdiff* value $\Delta(x_i, x_i')$ for each such application.

Unfold The `Unfold` combinator (Figure 4b) constructs a generative function $\mathcal{G}'$ that applies a kernel $\mathcal{G}_k$ repeatedly in sequence. `Unfold` is used to represent Markov chains, a common building block of probabilistic models, with state-space Y and transition kernel $\mathcal{G}_k$ from $\mathsf{Y} \times \mathsf{Z}$ to Y. The kernel $\mathcal{G}_k$ maps $(y, z) \in \mathsf{Y} \times \mathsf{Z}$ to a new state $y' \in \mathsf{Y}$. The generative function produced by this combinator takes as input the initial state $y_0 \in \mathsf{Y}$, parameters $z \in \mathsf{Z}$, and the number of Markov chain steps n to apply. Each kernel application is given an address namespace $i \in \{1 \ldots n\}$. Starting with initial state y_0, `Unfold` repeatedly applies $\mathcal{G}_k$ to each state and returns the vector of states $(y_1, \ldots, y_n)$. The custom GFI implementation in `Unfold` exploits the conditional independence of applications i and j given the return value of an intermediate application l, where $i < l < j$. The *retdiff* values returned by the kernel applications are used to determine which applications require a recursive call to the GFI.

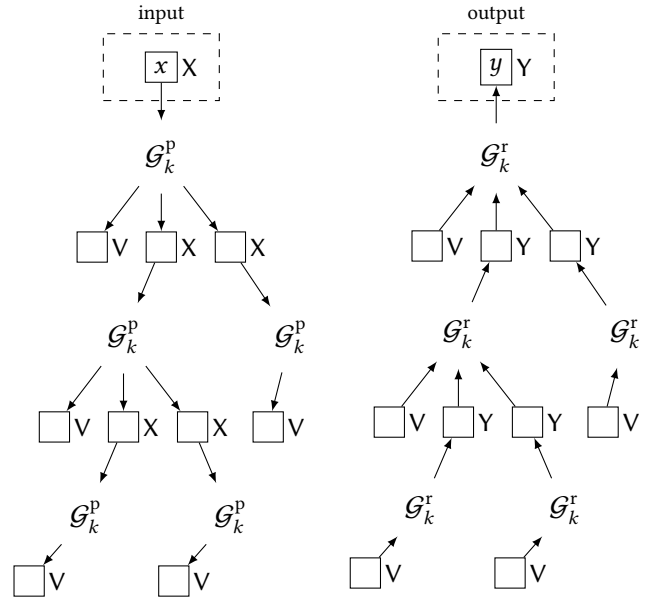
Recurse The `Recurse` combinator (Figure 4c) takes two kernels, a *production kernel* $\mathcal{G}_k^{\mathrm{p}}$ and a *reduction kernel* $\mathcal{G}_k^{\mathrm{r}}$, and returns a generative function $\mathcal{G}'$ that (i) recursively applies $\mathcal{G}_k^{\mathrm{p}}$ to produce a tree of values, then (ii) recursively applies $\mathcal{G}_k^{\mathrm{r}}$ to reduce this tree to a single return value. `Recurse` is used to implement recursive computation patterns including probabilistic context free grammars, which are a common building block of probabilistic models [17, 25, 33]. Letting b be the maximum branching factor of the tree, $\mathcal{G}_k^{\mathrm{p}}$ maps X to $\mathsf{V} \times (\mathsf{X} \cup \{\bot\})^b$ and $\mathcal{G}_k^{\mathrm{r}}$ maps $\mathsf{V} \times (\mathsf{Y} \cup \{\bot\})^b$ to Y, where $\bot$ indicates no child. Therefore, $\mathcal{G}'$ has input type X and return type Y. The `update` implementation uses *retdiff* values from the kernels to determine which subset of production and reduction applications require a recursive call to `update`.



(a) The `Map` generative function combinator.



(b) The `Unfold` generative function combinator.



(c) The `Recurse` generative function combinator. The left and right trees show production and reduction kernel applications, respectively.

Figure 4. A *generative function combinator* takes one or more generative functions called *kernels* and returns a generative function that exploits static patterns of conditional independence in its implementation of the generative function interface. Solid squares indicate the arguments and return values of kernel applications, and are annotated by their types. Dotted rectangles indicate the input arguments and output return value of the generative function produced by the combinator.

6 Implementation

This section describes an implementation of the above design, called Gen, that uses Julia as the host language, and includes three interoperable modeling languages embedded in Julia.

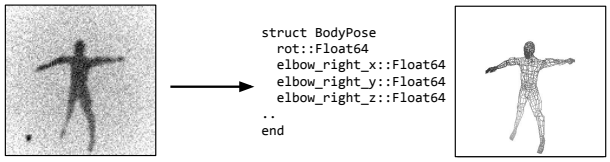
Dynamic Modeling Language (DML) The DML is shallowly embedded in Julia and uses dynamically computed addresses. The compiler produces generative functions whose traces are hash tables containing one entry for each generative function call and each random choice, where keys are relative addresses (Figure 2c). Each GFI method is implemented in Julia code that is effectively generated by applying a distinct transformational compiler to the body of the generative function. This is related to a standard probabilistic programming system architecture [66], except that in the standard architecture, transformational compilers produce generic inference algorithms and not primitives for composing inference algorithms. In the DML, reverse-mode automatic differentiation uses a tape of boxed values constructed with operator overloading. The internal proposal distribution is ancestral sampling [34].

Static Modeling Language (SML) The SML is a subset of the DML designed to support straightforward static analysis. Function bodies must be static single assignment basic blocks, and addresses must be literal symbols (`model` in Figure 2b is an example). These restrictions enable static inference of the address space of random choices, which enables specialized fast trace implementations; as well as static information flow analysis for efficient implementations of `update`. The compiler generates an information flow intermediate representation (IR) based on a directed acyclic graph with nodes for generative function calls, Julia expressions, and random choices. Traces are efficient Julia `struct` types that are generated statically for each function. A custom JIT compiler, implemented with Julia's multi-stage programming features, generates Julia code for each GFI method that is specialized to the generative function. For `update` the compiler uses the IR and the address schema (the set of top-level addresses) of the choice map u to identify statements that do not require re-evaluation. This avoids the runtime overhead of dynamic dependency tracking [40], while still exploiting fine-grained conditional independencies in basic blocks.

TensorFlow Modeling Language (TML) The TML encapsulates pure-functional TensorFlow (TF, [1]) computations as generative functions (example in Figure 5c). This permits scalable use of deep neural networks within models and proposal distributions. The TML consists of the TF Python API and `TFFunction`, which constructs a generative function from nodes in a pure-functional TensorFlow computation graph: (i) TF 'variables' are trainable parameters[2] of the generative

[2]This paper does not discuss *trainable parameters* of generative functions.

(a) 3D body pose inference task



(b) Model defined in the Dynamic Modeling Language (DML)

```
@gen function body_pose_prior()
  rot = @trace(uniform(0, 1), :rotation),
  elbow_r_x = @trace(uniform(0, 1), :elbow_right_x)
  elbow_r_y = @trace(uniform(0, 1), :elbow_right_y)
  elbow_r_z = @trace(uniform(0, 1), :elbow_right_z)
  ...
  pose = BodyPose(rot, elbow_r_x, elbow_r_y, elbow_r_z, ...)
  return pose
end

@gen function model()
  pose = @trace(body_pose_prior(), :pose)
  image = render_depth_image(pose)
  blurred = gaussian_blur(image)
  @trace(independent_pixel_noise(blurred, 0.1), :image)
end
```

(c) Neural net. defined in TensorFlow Modeling Language (TML)

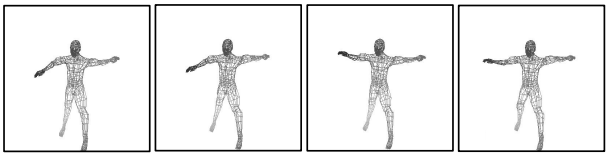
```
tf = pyimport("tensorflow")
image_flat = tf.placeholder(tf.float64)
image = tf.reshape(image_flat, [-1, 128, 128, 1])
W_conv1 = tf.Variable(initial_weight([5, 5, 1, 32]))
b_conv1 = tf.Variable(initial_bias([32]))
h_conv1 = tf.nn.relu(tf.add(conv2d(image, W_conv1), b_conv1))
h_pool1 = max_pool_2x2(h_conv1)
..
W_fc2 = tf.Variable(initial_weight([1024, 32]))
b_fc2 = tf.Variable(initial_bias([32]))
output = tf.add(tf.matmul(h_fc1, W_fc2), b_fc2)
neural_net = TFFunction([W_conv1, b_conv1, ..], [image_flat], output)
```

(d) Custom proposal in DML that invokes the neural network

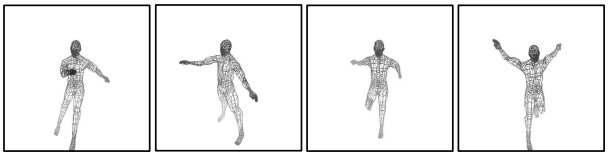
```
@gen function predict_body_pose((grad)(nn_output::Vector{Float64}))
  @trace(beta(exp(nn_output[1]), exp(nn_output[2])), :rotation)
  @trace(beta(exp(nn_output[3]), exp(nn_output[4])), :elbow_right_x)
  @trace(beta(exp(nn_output[5]), exp(nn_output[6])), :elbow_right_y)
  @trace(beta(exp(nn_output[7]), exp(nn_output[8])), :elbow_right_z)
..
end

@gen function proposal(image::Matrix{Float64})
  nn_input = reshape(image, 1, 128 * 128)
  nn_output = @trace(neural_net(nn_input), :network)
  @trace(predict_body_pose(nn_output[1,:]), :pose)
end
```

(e) Inference program for importance sampling with custom proposal

```
function inference_program(image::Matrix{Float64})
  observations = choicemap()
  observations[:image] = image
  (trace,) = importance_resampling(model, (), observations,
                proposal, (image,), 10) # use 10 particles
  return trace
end
```



(f) Samples from Gen inference program in (e) (1.3s / sample)



(g) Samples from importance sampling using Turing (1.3s / sample)

Figure 5. Code and evaluation for body pose inference. The model, written in the Dynamic Modeling Language (DML), renders a depth image from pose parameters with a graphics engine. The custom proposal combines the DML and the TensorFlow Modeling Language (TML) to pass an observed depth image through a neural network and propose pose parameters.

function, (ii) TF 'placeholders' are inputs to the function, and (iii) there is a node that defines the return value. Although TML functions do not make random choices, the GFI enables automatic differentiation to compose across TF computations and code written in other modeling languages.

Inference Library Gen's built-in *inference library* supports diverse inference algorithms, including importance sampling and Metropolis-Hastings using generic or custom proposal distributions, MAP optimization using gradients, training proposal distributions using amortized inference [59], particle filtering [15] including custom proposals and rejuvenation moves, MALA [52], Hamiltonian Monte Carlo [16], custom reversible jump MCMC samplers [24], and black box variational inference [50].

7 Evaluation

We evaluated Gen on a benchmark set of five challenging inference problems. For each problem, we compared the efficiency of inference algorithms implemented in Gen with the efficiency of implementations in other probabilistic programming systems. We compared Gen with Venture [40] (which introduced programmable inference), Edward [61] (which is built on TensorFlow), Turing [18] (which like Gen is embedded in Julia), Anglican [68] (which is embedded in Clojure), and Stan [8] (which statically compiles inference). The evaluation shows that Gen significantly outperforms Venture, Turing, and Anglican in all comparisons—often by multiple orders of magnitude. Stan and Edward can solve only one of the five benchmark problems, and Gen performs competitively with them on that problem. The efficiency gains in Gen derive from greater inference programming flexibility and performant system architecture. The Gen system and benchmarks are available at https://probcomp.github.io/Gen. The remainder of this section summarizes the results.

7.1 Robust Bayesian Regression

The first benchmark is a simple robust regression problem. The aim is to infer the slope and intercept from noisy linear data with several outliers. Table 1a shows a comparison of

Gen and Venture on an *uncollapsed* variant of the model that contains explicit discrete random choices which indicate whether each data point is an inlier or an outlier. Gen's implementation using the SML and the `Map` combinator gives a >200x speedup over Venture for two identical inference algorithms (Metropolis-Hastings with a custom proposal and gradient-based optimization). The incremental computation optimizations enabled by the SML and the `Map` combinator give a roughly 100x speedup over the DML implementation, which does not exploit incremental computation and scales as $O(n^2)$ where n is the number of observed data points (n is ~500 in the experiments). Table 1b shows a comparison of Gen with Stan, Edward, Anglican, and Venture on a collapsed variant of the model, where the discrete random choices for the outlier indicators are manually eliminated from the model by summing them out. For each system we measured the runtime needed to exceed an accuracy threshold on held-out data. Gen, Edward, and Stan (which cannot express the uncollapsed variant) give comparable results, all of which are 10x faster than Anglican and many orders of magnitude faster than Venture.

Table 1. Evaluation results for Bayesian robust regression.

	Inference Algorithm	Runtime (ms/step)
Gen (SML + `Map`)	Custom Metropolis Hastings	64ms (±1)
Gen (DML)	Custom Metropolis Hastings	7,376ms (±87)
Venture	Custom Metropolis Hastings	15,910ms (±500)
Gen (SML + `Map`)	Gradient-Based Optimization	74ms (±2)
Gen (DML)	Gradient-Based Optimization	7,384ms (±85)
Venture	Gradient-Based Optimization	17,702ms (±234)

(a) Measurements for inference in uncollapsed model. Runtimes are the time needed per step of the inference algorithm (± value in parentheses is symmetrized interquartile range across replicates).

	Inference Algorithm	Runtime (ms)
Stan	Hamiltonian Monte Carlo (NUTS)	53.4ms
Gen (SML + `Map`)	Gaussian Drift Metropolis Hastings	75.3ms
Edward	Hamiltonian Monte Carlo	76.6ms
Anglican	Gaussian Drift Metropolis Hastings	783ms
Venture	Gaussian Drift Metropolis Hastings	1.3×10^6ms

(b) Measurements for inference in collapsed model. Runtimes are the time needed to exceed a predictive likelihood threshold of −1290 nats (the approximate asymptote) on held-out test data.

7.2 Structure Learning for Gaussian Processes

We next consider inference in a state-of-the-art structure learning problem. The task is to infer the covariance function of a Gaussian process (GP) model of a time series data set, where the prior on covariance functions is defined using a probabilistic context free grammar (PCFG). The inferred covariance function can then be used to make forecast predictions, shown in the left panel of Figure 6. This task has been studied in machine learning [17] and probabilistic programming [54, 57]. The right panel of Figure 6 shows the runtime versus forecasting error achieved by three MCMC algorithms for this model implemented in Gen, which use different custom schedules of MCMC updates to the parse tree of the PCFG: (i) choosing a subtree to replace uniformly at random ('unbiased'); (ii) choosing a subtree to replace with probability that decreases exponentially in depth ('biased'); and (iii) always re-generating the entire parse tree ('full'). The flexibility of inference programming in Gen allows us to easily assess the time-accuracy trade-offs of these algorithms: The 'unbiased' and 'biased' schedules give comparable efficiency, and are more efficient than the 'full' schedule. We compared the runtime of the Gen implementation to Venture and hand-coded Julia implementations (all using the 'unbiased' schedule). We evaluate Gen implementations with and without Gen's `Recurse` combinator (Figure 4c), which automatically caches intermediate covariance matrix computations. The results (Table 2) show that Gen gives a 40x speedup over Venture, even without `Recurse`. The Gen implementation without `Recurse` performs comparably to the hand-coded Julia implementation. Using `Recurse` gives a 1.7x speedup.

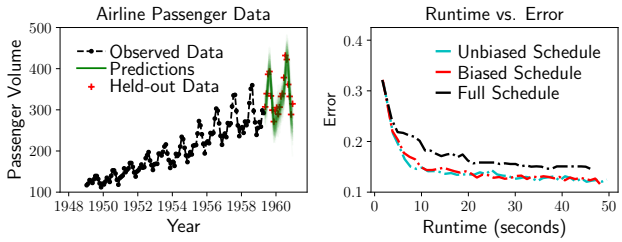


Figure 6. The left plot shows airline passenger volumes from 1948–1962, which exhibit linear and periodic trends. The right plot shows a comparison of runtime versus prediction error on held-out data, according to three inference algorithms for GP structure learning implemented in Gen.

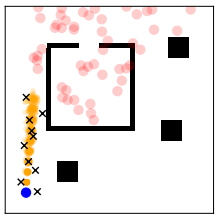
Table 2. Evaluation results for GP structure learning.

	Caching	Runtime (ms/step)
Gen (DML)	Provided by `Recurse`	2.57ms (± 0.09)
Julia (Handcoded)	None	4.73ms (± 0.45)
Gen (DML)	None	6.21ms (± 0.94)
Venture	None	279ms (± 31)

7.3 Algorithmic Model of an Autonomous Agent

We next consider MCMC inference in a model that is based on a simulator [12]. The model assumes a uniform prior distribution on the destination of an autonomous agent in a two-dimensional scene, and assumes that the agent moves from its starting location to its destination by following a path produced by a path planning algorithm based on rapidly exploring random trees (RRT, [37]). The task is to infer the destination of the agent from noisy measurements of its location over time. We evaluated two Gen implementations and

one Turing implementation of the same MCMC inference algorithm, which includes a cycle of three Metropolis-Hastings updates to: (i) the destination, (ii) the speed of the agent, and (iii) the measurement noise. All implementations used the same Julia path planner implementation, which makes thousands of random choices and dominates the running time of the algorithm, but is not instrumented[3] by Gen or Turing. The results (Figure 7) show that the Gen SML implementation outperforms the DML implementation by ~3x. This is because the SML uses static analysis of the model to avoid re-running the path planning algorithm when updating the speed and measurement noise variables. Both Gen implementations outperformed the Turing implementation.



	Runtime (sec.)
Gen (SML)	0.73s (± 0.01)
Gen (DML)	2.50s (± 0.08)
Turing	4.21s (± 0.13)

Figure 7. Left: Assumed start location of agent (blue), measured locations of agent over time (×) and samples of inferred location (orange) and destination (red). Polygons are obstacles. Right: Evaluation results for 1000 iterations of MCMC.

7.4 Nonlinear State-Space Model

We next consider particle filtering for object tracking in a nonlinear state-space model. An object is assumed to move along a piecewise-linear path at constant speed with Gaussian noise added to the distance traveled at each time step. The measurement model also assumes additive Gaussian noise. The task is to track the object over time along its assumed path. We evaluated two particle filtering inference algorithms implemented in Gen. The first uses a generic proposal distribution based on simulating the dynamics and the second uses a custom proposal derived by manual analysis of the model and expressed in Gen's DML. We compared these implementations to particle filtering implementations in Turing, Anglican, and Venture, none of which support custom particle filtering proposals. The results (Table 3) show that the custom proposal gives accurate results in an order of magnitude less time than the generic proposal. Furthermore, the Gen implementation using the generic proposal significantly outperforms the Anglican, Turing, and Venture implementations of the same algorithm.

7.5 Estimating 3D Body Pose from a Depth Image

We next consider a computer vision task [70]. The model (Figure 5) assumes a 3D articulated mesh model of a human body, parametrized by pose variables such as joint rotations, and a rendering process by which the mesh projects onto a depth image. The task is to infer the underlying 3D pose parameters from an observed depth image. We implemented two importance sampling (IS) algorithms: (i) a Turing implementation using a generic proposal (the prior), and (ii) a Gen implementation using a custom proposal that employs a deep neural network trained variationally on data simulated from the model. Neither Turing, Venture nor Anglican support custom IS proposals. The Gen proposal is a DML generative function that invokes a TML generative function. The Gen inferences are much more accurate than the Turing inferences for the same running time (Figures 5f and 5g).

[3]The random choices are not recorded in the trace. Gen's GFI formalizes *untraced* random choices, but this is outside the scope of this paper.

Table 3. Median runtime required to achieve a mean log marginal likelihood estimate above a threshold within two nats of a gold-standard estimate, for different implementations of particle filtering in a nonlinear state space model.

	Proposal Distribution	Runtime (ms)
Gen (DML + `Unfold`)	Custom	4.9ms (± 0.07)
Gen (DML + `Unfold`)	Generic	82ms (± 3.6)
Anglican	Generic	275ms (± 11)
Turing	Generic	1174ms (± 25)
Venture	Generic	$>10^6$ms

8 Related Work

Researchers have introduced many probabilistic programming systems over the last 20 years [8, 18, 22, 23, 40–42, 48, 49, 56, 61, 68]. We relate them to Gen along several axes.

System Architecture Gen's system architecture is a significant departure from existing work in probabilistic programming. Many recent systems [18, 23, 68] follow the sample/observe paradigm [63], in which probabilistic model code makes calls to `sample` and `observe` functions and is compiled to continuation-passing form. In the language backend, inference algorithms are implemented by intercepting program execution at each `sample` or `observe` checkpoint.

In Gen, there is no single notion of "probabilistic model code." Instead, Gen supports an extensible family of modeling languages and combinators, which produce as output *generative functions* (Julia objects that conform to the generative function interface). Because the inference library is implemented atop that interface, it can be used to perform inference on any generative function. This architectural design permits a high level of extensibility, with new modeling languages, compiler optimizations, inference algorithms, combinators, and more—all of which can be added in new libraries and modules without modifying Gen's system code.

Inference Implementation Efficiency For efficient implementation of MCMC or MAP updates that act on a subset of variables in a model, many probabilistic programming systems (including Gen) incrementally compute probability

ratios and trace updates [29, 40, 51, 69]. However, existing systems either lack Gen's modeling flexibility [29], are restricted to single-variable updates [51, 69], and/or have very high runtime overhead [40]. Gen's use of static analysis, JIT compilation, generative function combinators, and *argdiffs* provides a unique combination of implementation efficiency and flexibility of updates.

Customizing Inference Algorithms Most probabilistic programming systems, including Church [22], Stan [8], and Anglican [68], provide a fixed set of built-in, generic inference algorithms. This approach can be effective for restricted classes of problems, but generic algorithms frequently cannot satisfy the speed and accuracy requirements of real-world applications. Gen addresses this problem by building on Venture's *programmable inference* [40] approach, in which users specify custom inference algorithms by writing high-level *inference programs*. Systems besides Venture that provide some degree of inference programmability include WebPPL [23], Edward [61], Pyro [5], and Turing [18]. Inference programming techniques include Monte Carlo or variational *guide programs* [5, 23, 61], *drift kernels* for MCMC [3, 18, 23, 61] and scheduling inference tactics [18, 40, 61].

Gen offers significantly more affordances for customizing inference than other systems. In Venture, a fixed set of Monte Carlo proposal distributions and variational approximations are built into the system implementation. In Gen, users specify custom proposal distributions and variational approximations as generative functions. This is more expressive than Pyro [5]'s guide programs, which cannot be used with MCMC or sequential Monte Carlo (SMC). Custom inference techniques that are unique to Gen include custom Metropolis-Hastings (MH) proposals that update multiple variables at a time or make transdimensional moves (with automated acceptance probability calculation), incremental inference [10], applying inference tactics to arbitrary subsets of variables, and custom rejuvenation moves within SMC.

In Venture and Turing, inference programs are written in specialized domain-specific languages. In contrast, Gen inference programs are simply Julia programs that invoke inference library routines. This allows use of inference techniques not explicitly built into Gen's inference library. For example, Gen inference programs can initialize MCMC with an external numerical optimization routine or perform simulated annealing by following a temperature schedule even though Gen currently has no simulated annealing routine.

Coverage of Inference Algorithms Some systems, including Pyro and Edward, support both Monte Carlo and variational inference. However, Gen's support for Monte Carlo inference is significantly more flexible and complete than that of Pyro and Edward: Neither supports SMC, Pyro supports only generic MCMC algorithms like Hamiltonian Monte Carlo, and Edward's support for MH is not well developed for models with stochastic control flow. Gen's support for low-variance gradient estimates for scalable variational inference is currently less developed than that of Pyro and Edward. However, Gen's architecture makes it feasible to add advanced variational techniques without modifying Gen's modeling language implementations. Because Gen's inference library offers primitives for Monte Carlo and variational methods using the same abstractions, Gen inference programs can combine these methods to achieve the asymptotic accuracy of Monte Carlo inference and the efficiency of variational inference (e.g. see Section 7.5).

Statistical and Symbolic Optimizations One route to efficient inference algorithms is to automatically detect probabilistic relationships between model variables and perform meaning-preserving program transformations that simplify inference. In simple models, these techniques can even produce exact inference results. Systems that take this approach include Birch [45], which exploits conjugate prior relationships between model variables, and Hakaru [46], which can simplify algebraic expressions and symbolically disintegrate some models. But these systems have little or no support for custom inference programming, and in some cases, restrict the modeling language to make static analysis easier. A promising avenue of future research would be to support these techniques in a Gen modeling language that compiles to a generative function; this would enable these optimized models to be composed with other generative functions and used with Gen's entire inference library.

9 Conclusion

This paper describes Gen's flexible and extensible modeling and inference capabilities, and shows that Gen outperforms state-of-the-art probabilistic programming systems on problems such as object tracking, estimating 3D body pose from a depth image, and inferring the structure of a time series. Gen's extensibility offers new avenues for researchers and practitioners alike. For example, it is feasible to add embedded domain-specific modeling languages to Gen, each implementing the generative function interface, that capture problem structure from domains such as computer vision [35] and data modeling for databases [54]. It is also feasible to add inference languages that support verification [3]. Building DSLs on top of Gen allows for reuse of the inference library and reuse of tools for inspection and testing [11]. We hope the Gen system is a significant step towards making probabilistic programming suitable for general-purpose use.

Acknowledgments

This research was supported in part by the US Department of Defense through the the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, the DARPA SD2 program (contract FA8750-17-C-0239), and an anonymous philanthropic gift. We thank Martin Rinard and students of MIT Spring 2019 course 6.885 for helpful feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. USENIX Association, 265–283.

[2] Ryan Adams, Hanna Wallach, and Zoubin Ghahramani. 2010. Learning the structure of deep sparse graphical models. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*. PMLR, 1–8.

[3] Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying handcoded probabilistic inference procedures. (2018). arXiv:1805.01863

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Rev.* 59, 1 (2017), 65–98.

[5] 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. (2018). arXiv:1810.09538

[6] 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.

[7] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *Journal of Machine Learning Research* 3 (Jan. 2003), 993–1022.

[8] Bob Carpenter, Andrew Gelman, Matthew 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.

[9] Nick Chater, Joshua B. Tenenbaum, and Alan Yuille. 2006. Probabilistic models of cognition: Conceptual foundations. *Trends in Cognitive Sciences* 10, 7 (2006), 287–291.

[10] Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K. Mansinghka. 2018. Incremental inference for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 571–585.

[11] Marco Cusumano-Towner and Vikash K. Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. Curran Associates, Inc., 3000–3010.

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

[13] Martin de La Gorce, Nikos Paragios, and David J Fleet. 2008. Model-based hand tracking with texture, shading and self-occlusions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2008)*. IEEE, 1–8.

[14] Luca Del Pero, Joshua Bowdish, Daniel Fried, Bonnie Kermgard, Emily Hartley, and Kobus Barnard. 2012. Bayesian geometric modeling of indoor scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2012)*. IEEE, 2719–2726.

[15] Arnaud Doucet, Nando De Freitas, and Neil Gordon. 2001. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo Methods in Practice*, Arnaud Doucet, Nando de Freitas, and Neil Gordon (Eds.). Springer, 3–14.

[16] Simon Duane, Anthony D. Kennedy, Brian J. Pendleton, and Duncan Roweth. 1987. Hybrid Monte carlo. *Physics Letters B* 195, 2 (1987), 216–222.

[17] David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. 2013. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2010)*. PMLR, 1166–1174.

[18] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A language for flexible probabilistic inference. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS 2018)*. PMLR, 1682–1690.

[19] Alan E. Gelfand and Adrian F. M. Smith. 1990. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.* 85, 410 (1990), 398–409.

[20] Andrew Gelman, Hal S. Stern, John B. Carlin, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. CRC Press.

[21] Stuart Geman and Donald Geman. 1987. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. In *Readings in Computer Vision: Issues, Problem, Principles, and Paradigms*, Martin A. Fischler and Oscar Fischler (Eds.). Morgan Kaufmann Publishers Inc., 564–584.

[22] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2008)*. AUAI Press, 220–229.

[23] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2018-11-8.

[24] Peter J. Green. 1995. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika* 82, 4 (1995), 711–732.

[25] Roger B. Grosse, Ruslan Salakhutdinov, William T. Freeman, and Joshua B. Tenenbaum. 2012. Exploiting compositionality to explore a large space of model structures. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI 2012)*. AUAI Press, 306–315.

[26] Wilfred K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (1970), 97–109.

[27] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation* 18, 7 (2006), 1527–1554.

[28] Steven Holtzen, Yibiao Zhao, Tao Gao, Joshua B. Tenenbaum, and Song-Chun Zhu. 2016. Inferring human intent from video by sampling hierarchical plans. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1489–1496.

[29] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 111–125.

[30] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V. Gehler. 2015. The informed sampler. *Computing Vision and Image Understanding* 136, C (2015), 32–44.

[31] S. J. Julier and J. K. Uhlmann. 2004. Unscented filtering and nonlinear estimation. *Proc. IEEE* 92, 3 (2004), 401–422.

[32] Rudolph E. Kalman. 1960. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering* 82, 1 (1960), 35–45.

[33] Bjarne Knudsen and Jotun Hein. 2003. Pfold: RNA secondary structure prediction using stochastic context-free grammars. *Nucleic Acids Research* 31, 13 (2003), 3423–3428.

[34] Daphne Koller, Nir Friedman, and Francis Bach. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

[35] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. 2015. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2015)*. IEEE, 4390–4399.

[36] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning (ICML 2001)*. Morgan Kaufmann Publishers Inc., 282–289.

[37] Steven M. LaValle. 1998. *Rapidly-exploring random trees: A new tool for path planning*. Technical Report TR 98-11. Computer Science Department, Iowa State University.

[38] Vikash Mansinghka, Patrick Shafto, Eric Jonas, Cap Petschulat, Max Gasner, and Joshua B. Tenenbaum. 2016. CrossCat: A fully Bayesian nonparametric method for analyzing heterogeneous, high dimensional data. *Journal of Machine Learning Research* 17, 138 (2016), 1–49.

[39] Vikash K. Mansinghka, Tejas D. Kulkarni, Yura N. Perov, and Joshua B. Tenenbaum. 2013. Approximate Bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 1520–1528.

[40] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 603–616.

[41] Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22 (NIPS 2009)*. Curran Associates, 1249–1257.

[42] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic models with unknown objects. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*. Morgan Kaufmann Publishers Inc., 1352–1359.

[43] Michael Montemerlo, Sebastian Thrun, Daphne Roller, and Ben Wegbreit. 2003. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*. Morgan Kaufmann Publishers Inc., 1151–1156.

[44] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

[45] Lawrence M Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B Schön. 2018. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. (2018), 1037–1046.

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

[47] Radford M. Neal. 2000. Markov chain sampling methods for Dirichlet process mixture models. *Journal of Computational and Graphical Statistics* 9, 2 (2000), 249–265.

[48] Avi Pfeffer. 2001. IBAL: A probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, Vol. 1. Morgan Kaufmann Publishers Inc., 733–740.

[49] Avi Pfeffer. 2016. *Practical Probabilistic Programming* (1 ed.). Manning Publications Co.

[50] Rajesh Ranganath, Sean Gerrish, and David Blei. 2014. Black box variational inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS 2014)*. PMLR, 814–822.

[51] Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. 2016. C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS 2016)*. PMLR, 28–37.

[52] Gareth O. Roberts and Richard L. Tweedie. 1996. Exponential convergence of Langevin distributions and their discrete approximations. *Bernoulli* 2, 4 (December 1996), 341–363.

[53] Stuart J. Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach*. Pearson.

[54] Feras A. Saad, Marco Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 37 (2019), 29 pages.

[55] Tim Salimans, Diederik P Kingma, and Max Welling. 2015. Markov chain Monte Carlo and variational inference: Bridging the gap. In *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*. PMLR, 1218–1226.

[56] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[57] Ulrich Schaechtle, Feras Saad, Alexey Radul, and Vikash K. Mansinghka. 2016. Time Series Structure Discovery via Probabilistic Program Synthesis. (2016). arXiv:1611.07051

[58] Leonid Sigal, Alexandru Balan, and Michael J. Black. 2008. Combined discriminative and generative articulated pose and non-rigid shape estimation. In *Advances in Neural Information Processing Systems 20*. Curran Associates, Inc., 1337–1344.

[59] Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. 2013. Learning stochastic inverses. In *Advances in Neural Information Processing Systems 26 (NIPS 2013)*. Curran Associates, 3048–3056.

[60] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic Robotics*. MIT Press.

[61] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *International Conference on Learning Representations (ICLR)*.

[62] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele. 2014. Augur: Data-parallel probabilistic modeling. In *Advances in Neural Information Processing Systems 27 (NIPS 2014)*. 2600–2608.

[63] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. (2018). arXiv:1809.10756

[64] Martin J. Wainwright and Michael I. Jordan. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning* 1, 1 (2008), 1–305.

[65] Richard D. Wilkinson. 2013. Approximate Bayesian computation (ABC) gives exact results under the assumption of model error. *Statistical Applications in Genetics and Molecular Biology* 12, 2 (2013), 129–141.

[66] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14TH International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*. PMLR, 770–778.

[67] Frank Wood, Thomas L. Griffiths, and Zoubin Ghahramani. 2006. A non-parametric Bayesian method for inferring hidden causes. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006)*. AUAI Press, 536–543.

[68] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS 2014)*. PMLR, 1024–1032.

[69] Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. 2014. Generating efficient MCMC kernels from probabilistic programs. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS 2014)*. PMLR, 1068–1076.

[70] Mao Ye, Xianwang Wang, Ruigang Yang, Liu Ren, and Marc Pollefeys. 2011. Accurate 3D pose estimation from a single depth image. In *Proceedings of the International Conference on Computer Vision (ICCV 2011)*. IEEE, 731–738.