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

Marco F. Cusumano-Towner
Massachusetts Institute of Technology
marcoct@mit.edu

Feras A. Saad
Massachusetts Institute of Technology
fsaad@mit.edu

Alexander Lew
Massachusetts Institute of Technology
alexlew@mit.edu

Vikash K. Mansinghka
Massachusetts Institute of Technology
vkm@mit.edu

Abstract

Probabilistic modeling and inference are central to many fields. A key challenge for wider adoption of probabilistic programming languages is designing systems that are both flexible and performant. This paper introduces Gen, a new probabilistic programming system with novel language constructs for modeling and for end-user customization and optimization of inference. Gen makes it practical to write probabilistic programs that solve problems from multiple fields. Gen programs can combine generative models written in Julia, neural networks written in TensorFlow, and custom inference algorithms based on an extensible library of Monte Carlo and numerical optimization techniques. This paper also presents techniques that enable Gen's combination of flexibility and performance: (i) the generative function interface, an abstraction for encapsulating probabilistic and/or differentiable computations; (ii) domain-specific languages with custom compilers that strike different flexibility/performance tradeoffs; (iii) combinators that encode common patterns of conditional independence and repeated computation, enabling speedups from caching; and (iv) a standard inference library that supports custom proposal distributions also written as programs in Gen. This paper shows that Gen outperforms state-of-the-art probabilistic programming systems, sometimes by multiple orders of magnitude, on problems such as nonlinear state-space modeling, structure learning for real-world time series data, robust regression, and 3D body pose estimation from depth images.

1 Introduction

Probabilistic modeling and inference are central to diverse fields, such as computer vision, robotics, statistics, and artificial intelligence. Probabilistic programming languages aim to make it easier to apply probabilistic modeling and inference, by providing language constructs for specifying models and inference algorithms. Most languages provide automatic "black box" inference mechanisms based on Monte Carlo, gradient-based optimization, or neural networks. However, applied inference practitioners routinely customize an algorithm to the problem at hand to obtain acceptable performance. Recognizing this fact, some recently introduced languages offer "programmable inference" [29], which permits the user to tailor the inference algorithm based on the characteristics of the problem.

However, existing languages have limitations in flexibility and/or performance that inhibit their adoption across multiple applications domains. Some languages are designed to be well-suited for a specific domain, such as hierarchical Bayesian statistics (Stan [6]), deep generative modeling (Pyro [5]), or uncertainty quantification for scientific simulations (LibBi [33]). Each of these languages can solve problems in some domains, but cannot express models and inference algorithms needed to solve problems in other domains. For example, Stan cannot be applied to open-universe models, structure learning problems, or inverting software simulators for computer vision applications. Pyro and Turing [13] can represent these kinds of models, but exclude many important classes of inference algorithms. Venture [29] expresses a broader class of models and inference algorithms, but incurs high runtime overhead due to dynamic dependency tracking.

Key Challenges Two key challenges in designing a practical general-purpose probabilistic programming system are: (i) achieving good performance for heterogeneous probabilistic models that combine black box simulators, deep neural networks, and recursion; and (ii) providing users with abstractions that simplify the implementation of inference algorithms while being minimally restrictive.

This Work We introduce Gen, a probabilistic programming system that uses a novel approach in which (i) users define probabilistic models in one or more embedded *probabilistic DSLs* and (ii) users implement custom inference algorithms in the host language by writing *inference programs*

CSAIL Tech. Reports, 2018, MIT

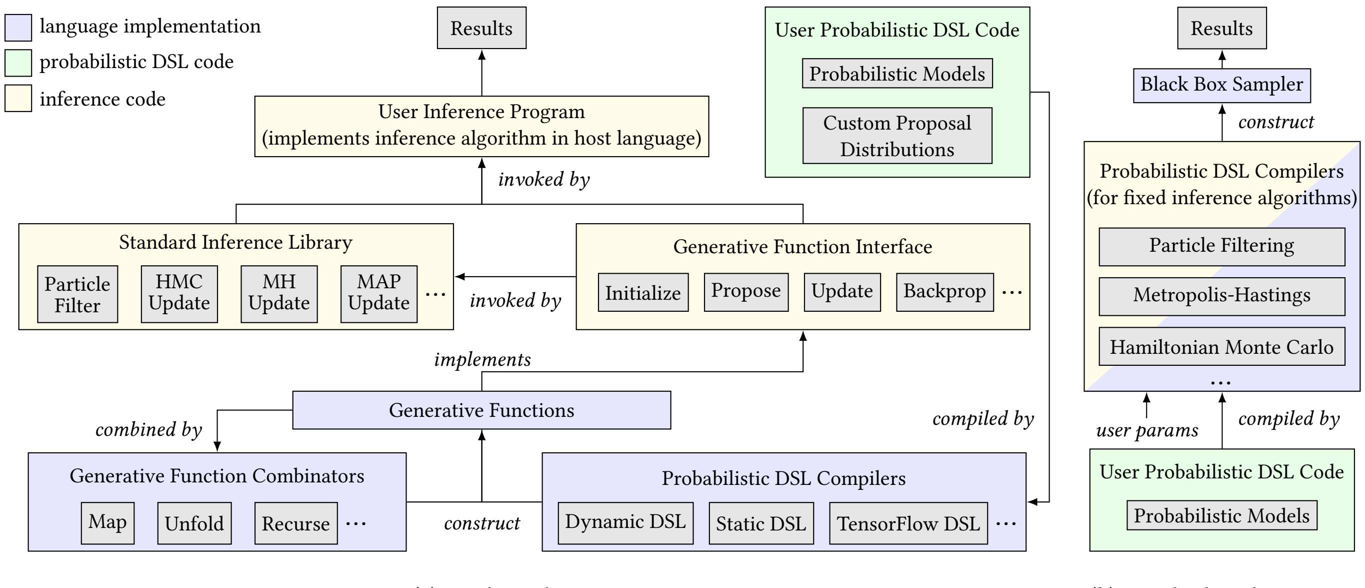


Figure 1. Comparison of Gen's architecture to a standard probabilistic programming architecture.

that manipulate the execution traces of models. This architecture affords users with modeling and inference flexibility that is important in practice. The Gen system, embedded in Julia [4], makes it practical to build models that combine structured probabilistic code with neural networks written in platforms such as TensorFlow. Gen also makes it practical to write inference programs that combine built-in operators for Monte Carlo inference and gradient-based optimization with custom algorithmic proposals and deep inference networks.

A Flexible Architecture for Modeling and Inference In existing probabilistic programming systems [13, 15, 48], inference algorithm implementations are intertwined with the implementation of compilers or interpreters for specific probabilistic DSLs that are used for modeling (Figure 1b). These inference algorithm implementations lack dimensions of flexibility that are routinely used by practitioners of probabilistic inference. In contrast, Gen's architecture (Figure 1a) abstracts away probabilistic DSLs and their implementation from inference algorithm implementation using the *generative function interface* (GFI). The GFI is a black box abstraction for probabilistic and /or differentiable computations that exposes several low-level operations on execution traces. *Generative functions*, which are produced by compiling probabilistic DSL code or by applying *generative function combinators* to other generative functions, implement the GFI. The GFI enables several types of domain-specific and problem-specific optimizations that are key for performance:

- Users can choose appropriate probabilistic DSLs for their domain and can combine different DSLs within the same model. This paper outlines two examples: a *TensorFlow DSL* that supports differentiable array computations resident on the GPU and interoperates with automatic differentiation in Julia, and a *Static DSL* that enables fast operations on execution traces via static analysis.
- Users can implement custom inference algorithms in the host language using the GFI and optionally draw upon a higher-level *standard inference library* that is also built on top of the GFI. User inference programs are not restricted by rigid algorithm implementations as in other systems.
- Users can express problem-specific knowledge by defining custom proposal distributions, which are essential for good performance, in probabilistic DSLs. Proposals can also be trained or tuned automatically.
- Users can easily extend the language by adding new probabilistic DSLs, combinators, and inference abstractions implemented in the host language.

Evaluation This paper includes an evaluation that shows that Gen can solve inference problems including 3D body pose estimation from a single depth image; robust regression; inferring the probable destination of a person or robot traversing its environment; and structure learning for real-world time series data. In each case, Gen outperforms existing probabilistic programming languages that support customizable inference (Venture and Turing), typically by one or more orders of magnitude. These performance gains are enabled by Gen's more flexible inference programming capabilities and high-performance probabilistic DSLs.

Main Contributions The contributions of this paper are:

- A probabilistic programming approach where users (i) define models in embedded probabilistic DSLs; and (ii) implement probabilistic inference algorithms using *inference* *programs* (written in the host language) that manipulate execution traces of models (Section 2).

- The *generative function interface*, a novel black box abstraction for probabilistic and/or differentiable computations that separates probabilistic DSLs from the implementation of inference algorithms (Section 3).
- *Generative function combinators*, which produce generative functions that exploit common patterns of conditional independence to enable efficient operations on execution traces. We give three examples of combinators: *map*, *unfold*, and *recurse* (Section 4).
- Gen, a system implementing this approach with three interoperable probabilistic DSLs (Section 5).
- An empirical evaluation of Gen's expressiveness and efficiency relative to other probabilistic programming languages on five challenging inference problems (Section 6).

2 Overview

In this section, we walk through an example probabilistic model, demonstrate three probabilistic inference programs for the model, and highlight the main features of Gen's architecture along the way. The example in Figure 2 is based on modeling a response variable y as a linear function of a dependent variable x. Figure 2a presents the inference problem: given a data set of observed pairs (x_i, y_i), the goal of probabilistic inference 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 presents code in a probabilistic domain-specific language that defines the probabilistic model. The three inference programs, implemented in Julia, are shown in Figure 2f. Each of these inference programs implements a different algorithm for solving the inference task and, as a result, exhibits different performance characteristics, shown in Figure 2g.

2.1 Example Probabilistic Model

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

Models are Generative Functions Users define probabilistic models by writing *generative functions* in *probabilistic DSLs*. Generative functions *generate* realizations of (typically stochastic) processes. The probabilistic DSL used in this example is called the *Dynamic DSL* (syntax in Figure 3). The model in this example is defined by the generative function '`model`' (Lines 14-29). Generative functions in the Dynamic DSL are identified by a `@gen` annotation in front of a regular Julia `function` definition. Dynamic DSL functions may use arbitrary Julia code, as long as they do not mutate any externally observable state. Dynamic DSL functions make random choices with functions like `normal` and `bernoulli` that sample from an extensible set of probability distributions (for instance, `normal(0,1)` draws a random value from the standard normal distribution) Each random choice represents either a latent (unknown) or observable quantity.

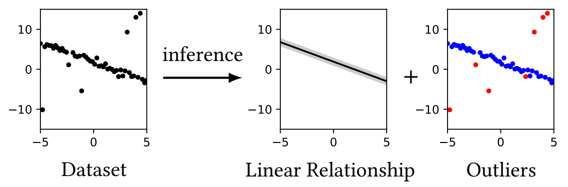
The generative function `model` takes as input a vector of dependent variables `xs` (containing the x_i) and returns a vector of response variables `ys` (containing the y_i). First, in Lines 15-18, the model makes random choices (shown in green) for the parameters (`prob_outlier`, `noise`, `slope`, `intercept`) that govern the linear relationship from their prior distributions. The parameters `slope` and `intercept` parameterize the assumed linear dependence of y_i on x_i. The parameter `noise` determines, for data points that are not outliers, the variance of the response variable around the straight regression line. Finally, the parameter `prob_outlier` is the probability that any given data point is an outlier.

Addresses of Random Choices Each random choice made when executing a generative function defined in the Dynamic DSL is given a unique *address* (shown in purple) using the `@addr` keyword. For example, the `prob_outlier` choice (Line 18) has address `:prob_outlier`. In general, the address of a choice is independent of any name to which its value may be assigned in the function body (note that random choices need not be bound to any identifier in the function body). Addresses are arbitrary Julia values that may be dynamically computed during function execution. The random choices seen so far have addresses that are Julia symbol values.

Generative Function Combinators Having generated the parameters, the generative function `model` next generates each y_i from the corresponding x_i, for each data point i (Lines 19-27). The `@diff` block (Lines 19-23, brown) is an optional performance optimization which we will discuss later. Lines 25-27 call the generative function '`generate_datum`' (Lines 1-10) once for each data point, passing in the x_i and each of the four parameters. To do this, we use `MapCombinator`, which is a *generative function combinator* (Section 4) that behaves similarly to the standard higher order function 'map'. On Line 12, the `MapCombinator` Julia function takes the generative function `generate_datum` and returns a second generative function (`generate_data`) that applies the first function repeatedly to a vector of argument values, using independent randomness for each application. On Line 25, the `model` function calls `generate_data`, which in turns calls `generate_datum` once for each data point i to generate the y_i from the corresponding x_i and the parameters. Within the ith application of `generate_datum`, a Bernoulli random choice (Line 4) determines whether data point i will be an outlier or not. If the data point is not an outlier (Line 7), then the response variable y_i has an expected value (`mu`) that is a linear function of x_i and standard deviation (`std`) equal to `noise`. If the data point is an outlier (Line 5), then y_i has an expected value of 0 and a large standard deviation (10). Finally, we sample the response variable y_i from a normal distribution with the given expected value and standard deviation (Line 9).

Hierarchical Address Spaces When calling other generative functions, `@addr` is used to indicate the *namespace*

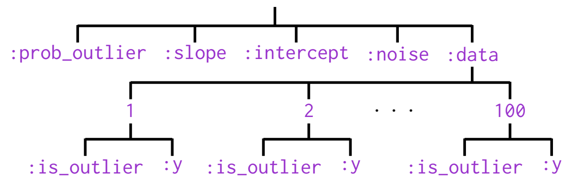
(a) Probabilistic inference task



(b) Probabilistic model

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

generate_data = MapCombinator(generate_datum)

@gen function model(xs::Vector{Float64})
  slope = @addr(normal(0, 2), :slope)
  intercept = @addr(normal(0, 2), :intercept)
  noise = @addr(gamma(1, 1), :noise)
  prob_outlier = @addr(uniform(0, 1), :prob_outlier)
  @diff begin
    addrs = [:prob_outlier, :slope, :intercept, :noise]
    diffs = [@choicediff(addr) for addr in addrs]
    argdiff = all(map(isnodiff,diffs)) ? noargdiff : unknownargdiff
  end
  n = length(xs)
  ys = @addr(generate_data(xs, fill(prob_outlier, n),
            fill(noise, n), fill(slope, n), fill(intercept,n)),
            :data, argdiff)
  return ys
end
```

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



(d) Custom proposals used by inference programs

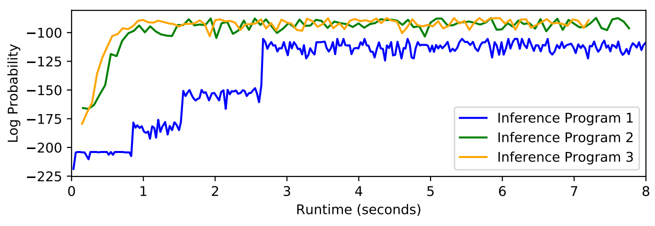
```
@gen function is_outlier_proposal(previous_trace, i::Int)
 is_outlier = previous_trace[:data => i => :is_outlier]
 @addr(bernoulli(is_outlier ? 0.0 : 1.0),
       :data => i => :is_outlier)
end

@gen function ransac_proposal(previous_trace, xs, ys)
  (slope, intercept) = ransac(xs, ys, 10, 3, 1.)
  @addr(normal(slope, 0.1), :slope)
  @addr(normal(intercept, 1.0), :intercept)
end
```

(e) Julia functions used in inference programs

```
function make_constraints(ys::Vector{Float64})
  constraints = Assignment()
  for i=1:length(ys)
    constraints[:data => i => :y] = ys[i]
  end
  return constraints
end

function parameter_update(trace, num_sweeps::Int)
  for j=1:num_sweeps
    trace = default_mh(model, select(:prob_outlier), trace)
    trace = default_mh(model, select(:noise), trace)
    trace = mala(model, select(:slope, :intercept), trace, 0.001)
  end
end

function is_outlier_update(trace, num_data::Int)
  for i=1:num_data
    trace = custom_mh(model, is_outlier_proposal, (i,), trace)
  end
  return trace
end
```

(f) Three user inference programs

```
function inference_program_1(xs::Vector{Float64}, ys::Vector{Float64})
    constraints = make_constraints(ys)
    (trace, _) = initialize(model, (xs,), constraints)
    for iter=1:100
        selection = select(:prob_outlier, :noise, :slope, :intercept)
        trace = default_mh(model, selection, (), trace)
        trace = is_outlier_update(trace, length(xs))
    end
    return trace
end

function inference_program_2(xs::Vector{Float64}, ys::Vector{Float64})
    constraints = make_constraints(ys)
    (slope, intercept) = least_squares(xs, ys)
    constraints[:slope] = slope
    constraints[:intercept] = intercept
    (trace, _) = initialize(model, (xs,), constraints)
    for iter=1:100
        trace = parameter_update(trace, 5)
        trace = is_outlier_update(trace, length(xs))
    end
    return trace
end

function inference_program_3(xs::Vector{Float64}, ys::Vector{Float64})
    constraints = make_constraints(ys)
    (trace, _) = initialize(model, (xs,), constraints)
    for iter=1:100
        trace = custom_mh(model, ransac_proposal, (xs, ys), trace)
        trace = parameter_update(trace, 5)
        trace = is_outlier_update(trace, length(xs))
    end
    return trace
end
```



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

Figure 2. Example of the proposed programming model. Probabilistic DSLs are used to define (b) probabilistic models and (d) custom proposal distributions. (e)–(f) algorithms are implemented by users in the host language, using methods provided by the standard inference library (shown in bold). (g) shows the time-accuracy profiles of the three inference programs in (f).

D := `normal` | `gamma` | `categorical` |...
E := J | R | F
R := `@addr`($D(E_2, E_3, ..), E_1$)
F := `@addr`($G(E_2, E_3, ..), E_1$)

Figure 3. Syntax of the Dynamic DSL, which is embedded in Julia. J are Julia expressions, G are names of generative functions, D are probability distributions, E are expressions, R are random choices, and F are calls to generative functions.

relative to which all random choices made during the call are addressed. For example, `model` calls `generate_data` with namespace `:data` (Lines 25-27). Generative functions that are produced by generative function 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). Figure 2c presents the hierarchical address space of random choices made by `model`. Hierarchical addresses are constructed by chaining together address components using the => operator. For example, `:data => 2 => :is_outlier` refers to the `is_outlier` random choice for the second data point.

Argdiffs Enable Efficient Trace Updates The `@diff` code block in `model` (Lines 19-23, brown) is optional, but important for good performance of Markov Chain Monte Carlo (MCMC) inference. MCMC inference often involves making small changes to an execution of a generative function. The `@diff` block is a feature of the Dynamic DSL that allows users to insert code that computes the change to the arguments to a call (the *argdiff*) from one execution to another execution, as well as the change to the return value of a function (the *retdiff*, not used in this example). The *argdiff* can indicate that the arguments to a call have not changed (`noargdiff`) or that they may have changed (`unknownargdiff`), or can provide detailed information about the change. In this case, the *argdiff* passed to the call to `generate_data` indicates whether or not the parameters of the model have changed from the previous execution. If the parameters have changed, then the call to `generate_datum` must be visited for each data point to construct the new execution trace. If the parameters have not changed, then most calls to `generate_datum` can be avoided. This feature enables efficient asymptotic scaling of MCMC. Note that code in the `@diff` block and the `argdiff` variable (Line 27) are not part of the generative function body and are only used when updating an execution.

2.2 Example User Inference Programs

Figure 2f shows three user inference programs, which are Julia functions that implement different custom algorithms for inference in the example probabilistic model. These inference programs are implemented using high-level abstractions provide by our system (these methods e.g. **`select`** and **`initialize`** are shown bold-face in the code figures). Each inference program takes as input a data set containing many (x_i, y_i) values; it returns inferred values for the parameters that govern the linear relationship and the outlier classifications.

Execution Traces as Data Structures 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') also contains the arguments on which the function was executed, and the function's return value. Traces are persistent data structures.

Generative Function Interface Each inference program in Figure 2f first obtains an initial trace by calling the **`initialize`** 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 (Lines 65, 79, 89). The **`initialize`** method is part of the *generative function interface* (GFI, Section 3), which exposes low-level operations on execution traces. The `constraints` are an *assignment* (produced by Figure 2e, Line 41), which is a prefix tree that maps addresses to values. Users construct assignments that map addresses of observed random choices to their observed values. Each inference program constrains the y-coordinates to their observed values. Inference program 2 also constrains the slope and intercept (Lines 77-78) to values obtained from a heuristic procedure (`least_squares`) implemented in Julia. The other two inference programs initialize the slope and intercept to values randomly sampled from the prior distribution.

Standard Inference Library After obtaining an initial trace, each program in Figure 2f then uses a Julia `for` loop (Lines 66, 80, 90) to repeatedly apply a cycle of different Markov Chain Monte Carlo (MCMC) updates to various random choices in the trace. These updates improve the random choices in the initial trace in accordance with the constrained observations. The MCMC updates use the methods **`default_mh`** and **`custom_mh`** from the *standard inference library*, which provides higher-level building blocks for inference algorithms that are built on top of the GFI. Inference program 1 uses **`default_mh`**, which proposes new values for certain selected random choices according to a default proposal distribution and accepts or rejects according to the Metropolis-Hastings (MH, [19]) rule.

Custom Proposals are Generative Functions Each inference program in Figure 2f updates the outlier indicator variables using `is_outlier_update` (Figure 2e, Lines 57-62). This procedure loops over the data points and performs an MH update to each `:is_outlier` choice using the **`custom_mh`** method from the standard inference library. The update uses a custom proposal distribution defined by `is_outlier_proposal` (Figure 2d, Lines 30-34) 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 defined as generative functions using the same probabilistic DSLs used to define probabilistic models. Inference program 3 uses a different MH update with a custom proposal (`ransac_proposal`, Figure 2d Lines 36-40) to propose new values for the slope and intercept by adding noise to values estimated by the RANSAC [12] robust estimation algorithm.

Using Host Language Abstraction Inference programs 2 and 3 in Figure 2d both invoke `parameter_update` (Figure 2e, Lines 49-55), which applies MH updates with default proposals to `:prob_outlier` and `:noise`, but uses a Metropolis-Adjusted Langevin Algorithm (MALA, [39]) update for the slope and intercept (**`mala`**, Line 53). The standard inference library method **`mala`** relies on automatic differentiation in the model function (`@ad` annotation, Lines 2-3) to compute gradients of the log probability density with respect to the selected random choices. Inference programs make use of familiar host language constructs, including procedural abstraction (e.g. reusable user-defined inference sub-routines like `parameter_update`) and loop constructs (to repeatedly invoke nested MCMC and optimization updates). These host language features obviate the need for new and potentially more restrictive 'inference DSL' constructs [13, 28]. Using the host language for inference programming also allows for seamless integration of external code into inference algorithms, as illustrated by the custom least squares initialization in inference program 1 (Line 76) and the custom external RANSAC estimator used in inference program 3 (Line 91).

Performance Depends on the Inference Algorithm As seen in Figure 2g, the three inference programs perform differently. The default proposal for the parameters used in inference program 1 causes slow convergence as compared to the combination of MALA updates and custom initialization used in inference program 2. The custom RANSAC proposal used in inference program 3 also improves convergence—it does not employ local random walk search strategy like the MALA update, but instead proposes approximate estimates from scratch with each application. The constructs in our system make it practical for the user to experiment with and compare various inference strategies for a fixed model.

3 Generative Function Interface

The *generative function interface* (GFI) is a black box abstraction for probabilistic and/or differentiable computations that provides an abstraction barrier between the implementation of probabilistic DSLs and implementations of inference algorithms. *Generative functions* are objects that implement the methods in the GFI and are typically produced from DSL code by a probabilistic DSL compiler. For example, the Dynamic DSL compiler takes a `@gen` function definition and produces a generative function that implements the GFI. Each method in the GFI performs an operation involving execution traces of the generative function on which it is called.

3.1 Formalizing Generative Functions

We now formally describe generative functions.

Assignment An *assignment* 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 $\mathrm{dom}[s] := \{a \in \mathsf{A} \mid s(a) \neq \bot\}$ of s as the set of non-vacuous addresses in s. Let $s|_B$ be the assignment obtained by restricting s to a subset of addresses $B \subseteq \mathrm{dom}[s]$, i.e. $s|_B(a) = s(a)$ if $a \in B$ and $s|_B(a) = \bot$ otherwise. For two assignments 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 $\mathsf{S} := \mathsf{V}^{\mathsf{A}}$ be the set of all assignments 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 assignments $s \in \mathsf{S}$ indexed by argument $x \in \mathsf{X}$. In words, x is a fixed argument to the generative function and s is a realization of all the random choices made during the execution. Since p is a distribution, for each x, the series $\sum_{s \in \mathsf{S}} p(x, s) = 1$. We use the notation $p(s; x) := p(x, s)$ to separate the arguments x from the assignment s. The output function $f : X \times \{s : p(s; x) > 0\} \to Y$ maps an argument x and an assignment s of all random choices (whose probability under p is non-zero) to a return value $f(x, s) \in \mathsf{Y}$. Finally, $q : \mathsf{X} \times \mathsf{S} \times \mathsf{S} \to [0, 1]$ is an internal proposal distribution that assigns a probability $q(s; x, u)$ to each assignment s for all $x \in \mathsf{X}$ and all $u \in \mathsf{S}$. The proposal distribution q satisfies (i) $\sum_s q(s; x, u) = 1$ for all $x \in \mathsf{X}$ and $u \in \mathsf{S}$; and (ii) $q(s; x, u) > 0$ if and only if $s \cong u$ and $p(s; x) > 0$.

3.2 Interface Methods

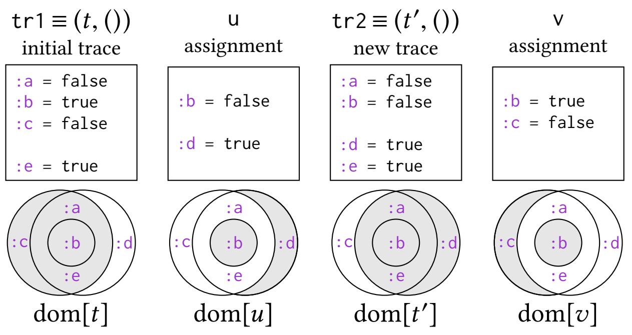
We now describe several methods in the GFI. Each method takes a generative function $\mathcal{G}$ in addition to other method-specific arguments. We will denote an assignment by t if $p(t; x) > 0$ (for some $x \in \mathsf{X}$) when the assignment is 'complete' and by u when the assignment is 'partial'. An *execution trace* (t, x) is a pair of a complete assignment t and argument x.

`initialize` ***(sampling an initial trace)*** This method takes an assignment 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 $w := p(t; x)/q(t; x, u)$. It is an error if no such assignment t exists.

propose ***(proposing an assignment)*** This method samples an assignment from a generative function $\mathcal{G}$ that is used as a proposal distribution and computes (or estimates) the proposal probability. In particular, `propose` takes arguments x and a set of addresses B, and returns an assignment u and a weight w by (i) sampling $t \sim p(\cdot; x)$; then (ii) setting $u := t|_B$ and $w := p(t; x)/q(t; x, u)$. If $B = \varnothing$ (i.e. no addresses are provided) then the method sets $u := t$.

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

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

# Sample a random initial trace (tr1).
x = ()                           # empty tuple, foo takes no args
u = Assignment()                 # empty assignment
(tr1, w1) = initialize(foo, x, u) # 'tr1' is the trace

# Use 'update' to obtain a trace (tr2) with :b=false, :d=true.
u =  Assignment([(:b, false), (:d, true)])
(tr2, w2, v, retdiff) = update(foo, x, noargdiff, tr1, 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 4. Illustration of `update` on 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, the `update` method takes a trace (t, x) of $\mathcal{G}$; new arguments x'; and a partial assignment u. It returns a new trace (t', x') such that $t' \cong u$ and $t'(a) = t(a)$ for $a \in (\mathrm{dom}[t] \cap \mathrm{dom}[t']) \setminus \mathrm{dom}[u]$. That is, values are copied from t for addresses that remain and that are not overwritten by u. It is an error if no such t' exists. The method also returns a *discard assignment* v where $v(a) := t(a)$ for $a \in (\mathrm{dom}[t] \setminus \mathrm{dom}[t']) \cup (\mathrm{dom}[t] \cap \mathrm{dom}[u])$. 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 4 illustrates the behavior of `update` in a simple setting.

Oftentimes, in an invocation of `update` the assignment 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 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). The types of *argdiff* and *retdiff* values depend on the generative function $\mathcal{G}$. Section 2 shows an example usage of *argdiff*.

backprop ***(automatic differentiation)*** This method computes gradients of the log probability of the execution trace with respect to the values of function arguments x and/or the values of random choices made by $\mathcal{G}$. In particular, `backprop` takes a trace (t, x) and a selection of addresses B that identify the random choices 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. `backprop` also accepts an optional return value gradient $\nabla_y J$ (for some J) in which case it returns $\nabla_B(\log p(t; x) + J)$ and $\nabla_x(\log p(t; x) + J)$. Note that for $\nabla_B \log p(t; x)$ to be defined, random choices in A must have differentiable density functions with respect to a base measure, in which case $\log p(t; x)$ is a joint density with respect to the induced base measure on assignments which is derived from the base measures for each random choice.

3.3 Implementing a Metropolis-Hastings Update

A key contribution of the GFI is its ability to act as the building block for implementing many probabilistic inference algorithms. We next present an example showing how to use the GFI to implement the Metropolis-Hastings update (`custom_mh`) used in Figure 2e. We call `propose` on the proposal function (`proposal`) to generate the proposed assignment (`u`) and the forward proposal probability (`fwd_score`). The `update` method takes the previous model trace (`trace`) and the proposed assignment, and returns (i) a new trace that is compatible with the proposed assignment; (ii) the model's contribution (`w`) to the acceptance ratio; and (iii) the discard assignment (`v`), which would have to be proposed to reverse the move. Then, `assess` is used to compute the reverse proposal probability (`rev_score`). Finally, `mh_accept_reject` stochastically accepts the move with probability `alpha`; if the move is accepted, the new trace is returned, otherwise the previous trace is returned. (Note that the weight `w` returned by the GFI is in log-space).

```
function custom_mh(model, proposal, proposal_args, trace)
  proposal_args_fwd = (trace, proposal_args...,)
  (u, fwd_score) = propose(proposal, proposal_args_fwd)
  args = get_args(trace) # model arguments
  (new_trace, w, v, _) = update(model, args, noargdiff, trace, u)
  proposal_args_rev = (new_trace, proposal_args...,)
  rev_score = assess(proposal, proposal_args_rev, v)
  alpha = w + rev_score - fwd_score
  return mh_accept_reject(alpha, trace, new_trace)
end
```

3.4 Compositional Implementation Strategy

Each GFI method is designed to be implemented compositionally by invoking the same GFI method for each generative function call within a generative function. In this implementation strategy, execution trace data structures are hierarchical and have a 'sub-trace' for each generative function call. Therefore, when a probabilistic DSL compiler produces a generative function $\mathcal{G}$ from probabilistic DSL code, it can treat any generative function that is called by $\mathcal{G}$ as a black box that implements the GFI. This compositional design enables (i) generative functions defined using independent probabilistic DSLs to invoke one another and (ii) powerful extensions with new types of generative functions, as we demonstrate in the next two sections.

4 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 a call to `custom_mh` with `is_outlier_proposal` in Figure 2e, Line 59. This call proposes a change to the `:is_outlier` choice for a single data point (`i`) and then invokes `update` on the model function. Because data points are conditionally independent given the parameters, an efficient implementation that exploits conditional independence can 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 address this issue, 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.

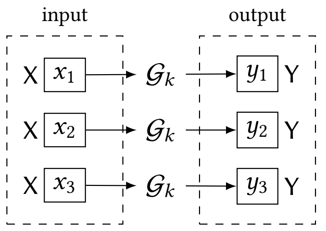
Map The *map* combinator (Figure 5a) constructs a generative function $\mathcal{G}'$ that independently applies a kernel $\mathcal{G}_k$ from X to Y to each element of a vector of inputs $(x_1, \dots, x_n)$ and returns a vector of outputs $(y_1, \dots, y_n)$. Each application of $\mathcal{G}_k$ is assigned an address namespace $i \in \{1 \dots n\}$. The implementation of `update` for *map* only makes recursive GFI calls for the kernel on those applications i for which the assignment 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 5b) constructs a generative function $\mathcal{G}'$ that applies a kernel $\mathcal{G}_k$ repeatedly in sequence. *Unfold* is used to represent a Markov chain, 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 $y_0 \in \mathsf{Y}$, the parameters $z \in \mathsf{Z}$ of $\mathcal{G}_k$, and the number of Markov chain steps n to apply. Each kernel application is given an address namespace $i \in \{1 \dots n\}$. Starting with initial state y_0, *unfold* repeatedly applies $\mathcal{G}_k$ to each state and returns the vector of states $(y_1, \dots, y_n)$. 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.

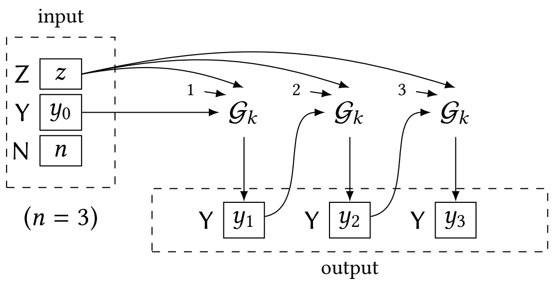
Recurse The *recurse* combinator (Figure 5c) takes two kernels, a *production kernel* $\mathcal{G}_k^{\mathrm{p}}$ and a *reduction kernel* $\mathcal{G}_k^{\mathrm{r}}$, and returns a generative function $\mathcal{G}'$ that (i) recursively applies $\mathcal{G}_k^{\mathrm{p}}$ to produce a tree of values, then (ii) recursively applies the $\mathcal{G}_k^{\mathrm{r}}$ 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 [11, 18, 22, 46]. Letting b be the maximum branching factor of the tree, $\mathcal{G}_k^{\mathrm{p}}$ maps X to $\mathsf{V} \times (X \cup \{\perp\})^b$ and $\mathcal{G}_k^{\mathrm{r}}$ maps $\mathsf{V} \times (\mathsf{Y} \cup \{\perp\})^b$ to Y, where $\perp$ 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`.

5 Implementation

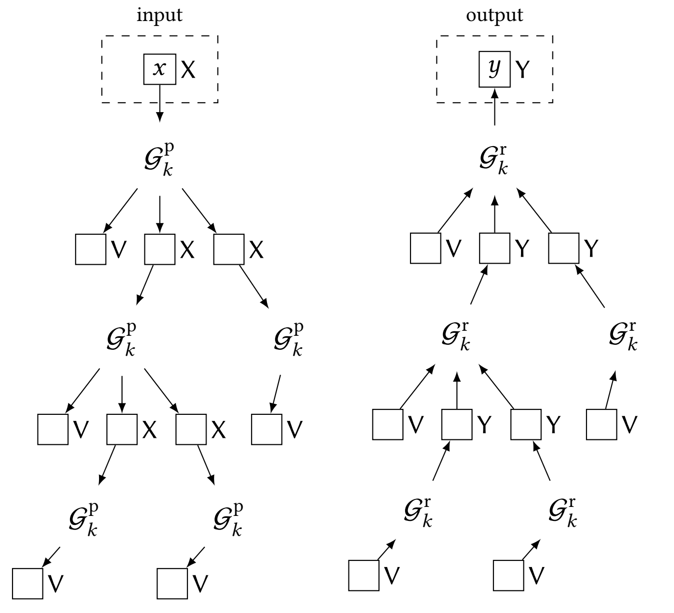
This section describes an implementation of the above design, called Gen, that uses Julia as the host language, and includes three interoperable probabilistic DSLs embedded in Julia.



(a) The *map* generative function combinator.



(b) The *unfold* generative function combinator.



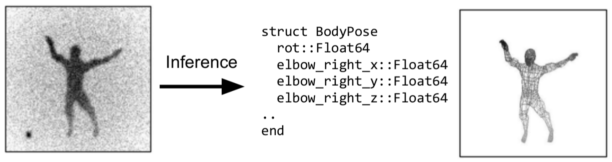
(c) The *recurse* generative function combinator. The left and right trees show production and reduction kernel applications, respectively.

Figure 5. 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.

Dynamic DSL The compiler for the Dynamic DSL in Figure 3 produces generative functions whose traces are hash tables that contain one entry for each generative function call and random choice, keyed by the address relative to the caller's namespace (Figure 2c). Each GFI method is implemented by applying a different transformational compiler to the body of the generative function, to generate a Julia function that implements the GFI method. Different GFI methods replace `@addr` expressions with different Julia code that implements the method's behavior. This is closely related to the architecture of other probabilistic programming systems [13, 15, 47, 48] except that in existing systems, transformational compilers produce implementations of *inference algorithms* and not primitives for composing many inference algorithms. In the Dynamic DSL, reverse-mode automatic differentiation uses a dynamic tape of boxed values constructed using operator overloading. Default proposal distributions use ancestral sampling [23].

Static DSL The *Static DSL* is a subset of the Dynamic DSL in which function bodies are restricted to be static single assignment basic blocks, and addresses must be literal symbols (see Figure 6 for examples, identified by the `@static` keyword). 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 that enables efficient implementations of `update`. The compiler generates an information flow intermediate representation based on a directed acyclic graph with nodes for generative function calls, Julia expressions, and random choices. Traces are Julia `struct` types, which are generated statically for each function, and are more efficient than the generic trace data structures used by the Dynamic DSL. A custom JIT compiler, implemented using Julia's *generated* function multi-stage programming feature, generates Julia code for each GFI method that is specialized to the generative function. The JIT compiler for `update` uses the intermediate representation and the address schema (the set of top-level addresses) of the assignment u to identify statements that do not require re-evaluation. For JIT compilation based on assignment schemata, the Julia type of an assignment includes its address schema. This avoids the runtime overhead of dynamic dependency tracking [28], while still exploiting fine-grained conditional independencies in basic blocks.

TensorFlow DSL The *TensorFlow DSL* is expresses functional TensorFlow (TF, [1]) computations (see Figure 6 for an example, identified by the `@tensorflow_function` keyword). This DSL allows for scalable use of deep neural networks within probabilistic models and proposal distributions. The DSL includes declarations for function inputs (`@input`, corresponding to TF 'placeholders'), trainable parameters (`@param`, corresponding to TF 'variables'), and return values (`@output`);



```
struct BodyPose
  rot::Float64
  elbow_right_x::Float64
  elbow_right_y::Float64
  elbow_right_z::Float64
..
end
```

(a) 3D body pose inference task

(b) Probabilistic model defined in the Static DSL

```
@static @gen function body_pose_prior()
  rot::Float64 = @addr(uniform(0, 1), :rotation),
  elbow_r_x::Float64 = @addr(uniform(0, 1), :elbow_right_x),
  elbow_r_y::Float64 = @addr(uniform(0, 1), :elbow_right_y),
  elbow_r_z::Float64 = @addr(uniform(0, 1), :elbow_right_z),
  ...
  return BodyPose(rot, elbow_r_x, elbow_r_y, elbow_r_z, ...)
end

@static @gen function model()
  pose::BodyPose = @addr(body_pose_prior(), :pose)
  image::Matrix{Float64} = render_depth_image(pose)
  blurred::Matrix{Float64} = gaussian_blur(image, 1)
  @addr(pixel_noise(blurred, 0.1), :image)
end
```

(c) Neural network defined in the TensorFlow DSL

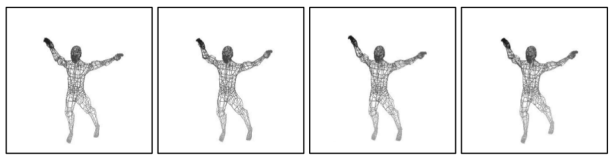
```
neural_network = @tensorflow_function begin
  @input image_flat Float32 [-1, 128 * 128]
  image = tf.reshape(image_flat, [-1, 128, 128, 1])
  @param W_conv1 initial_weight([5, 5, 1, 32])
  @param b_conv1 initial_bias([32])
  h_conv1 = tf.nn.relu(conv2d(image, W_conv1) + b_conv1)
  h_pool1 = max_pool_2x2(h_conv1)
  ...
  @param W_fc2 initial_weight([1024, 32])
  @param b_fc2 initial_bias([32])
  @output Float32 (tf.matmul(h_fc1, W_fc2) + b_fc2)
end
```

(d) Custom proposal that invokes the neural network

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

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

(e) Inference program for importance sampling with custom proposal

```
function inference_program(image::Matrix{Float64})
  observations = Assignment()
  observations[:image] = image
  (model_traces, weights) = custom_importance(model, (), proposal,
    (image,), observations=observations, num_particles=100)
  return (model_traces, weights)
end
```


(f) Samples from inference program using Gen (e) (5s / sample)

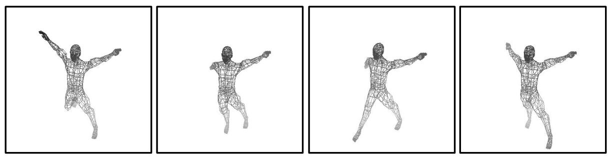

(g) Samples from importance sampling using Turing (90s / sample)

Figure 6. Modeling and inference code, and evaluation results for body pose inference task. The model, which is written in the Static DSL, invokes a graphics engine to render a depth image from pose parameters. The custom proposal combines the Static DSL and the TensorFlow DSL to pass an observed depth image through a deep neural network and propose pose parameters.

and statements that define the TF computation graph. Functions written in this DSL do not make random choices, but they do exercise the automatic differentiation (AD) methods in the GFI, which enable AD to compose across the boundary between TensorFlow computations and code written in other probabilistic DSLs. Trainable parameters of TensorFlow functions are managed by the TensorFlow runtime. We use a Julia wrapper [27] around the TensorFlow C API.

Standard Inference Library Gen's *standard inference library* includes support for diverse inference algorithms, and building blocks of inference algorithms, including importance sampling [37] and Metropolis-Hastings MCMC [19] using default proposal and custom proposal distributions, maximum-a-posteriori optimization using gradients, training proposal distributions using amortized inference [20, 26, 43], particle filtering [9] including custom proposals and custom rejuvenation kernels, Metropolis-Adjusted Langevin Algorithm [39], Hamiltonian Monte Carlo [10], and custom reversible jump MCMC samplers [16]. The implementation supports auxiliary variable methods like particle MCMC [2] with user-defined generative function implementations.

6 Evaluation

We evaluated Gen on a benchmark set of five challenging inference problems: (i) robust regression; (ii) inferring the probable destination of a person or robot traversing its environment; (iii) filtering in a nonlinear state-space model; (iv) structure learning for real-world time series data; and (v) 3D body pose estimation from a single depth image. For each problem, we compared the performance of inference algorithms implemented in Gen with the performance of implementations in other probabilistic programming systems. In particular, we compare Gen with Venture [28] (which introduced programmable inference), Turing [13] (which like Gen is embedded in Julia), and Stan [6] (which uses a black-box inference algorithm). Our evaluations show that Gen significantly outperforms Venture and Turing on all inference problems—often by multiple orders of magnitude. Stan can solve only one of the five benchmark problems, and Gen performs competitively with Stan on that problem. The performance gains in Gen are due to a combination of greater inference programming flexibility and more performant system architecture. We now summarize the results.

6.1 Robust Bayesian Regression

We first consider a model for robust Bayesian regression. Inference in robust Bayesian models is an active research area [45]. We evaluated Gen, Venture, and Stan implementations of MCMC and optimization-based inference algorithms in two variants of this model—a more difficult *uncollapsed* variant and an easier *collapsed* variant, in which a manual semantics-preserving program transformation removes certain random choices. Stan's modeling language cannot express the uncollapsed variant. The results in Table 1 indicate that Gen's performant implementation of the Static DSL gives 200x–700x speedup over Venture and comparable runtime to Stan. The optimizations enabled by the Static DSL give up to 10x speedup over the Dynamic DSL.

Table 1. Evaluation results for robust regression.

	Inference Algorithm	Runtime (ms/step)
Gen (Static)	MH Sampling	55ms (±1)
Gen (Static)	Gradient-Based Optimization	66ms (±2)
Gen (Dynamic)	Gradient-Based Optimization	435ms (±12)
Gen (Dynamic)	MH Sampling	510ms (±19)
Venture	MH Sampling	15,910ms (±500)
Venture	Gradient-Based Optimization	17,702ms (±234)

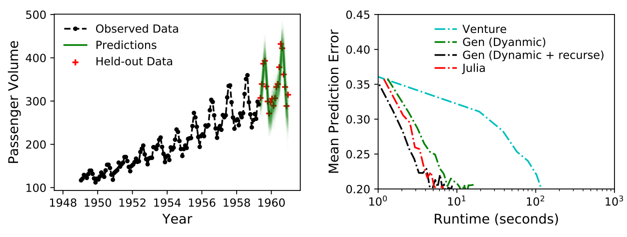
(a) Runtime measurements for inference in uncollapsed model

	Inference Algorithm	Runtime (ms/step)
Gen (Static)	MH Sampling	4.45ms (±0.49)
Stan	Gradient-Based Sampling	5.26ms (±0.38)
Gen (Static)	Gradient-Based Sampling	18.29ms (±0.24)
Gen (Static)	Gradient-Based Optimization	37.14ms (±0.93)
Venture	MH Sampling	3,202ms (±17)
Venture	Gradient-Based Optimization	8,159ms (±284)

(b) Runtime measurements for inference in collapsed model

6.2 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 Figure 7a. This task has been studied in machine learning [11] and probabilistic programming [29, 40, 42]. We evaluated Gen, Venture, and Julia implementations of an MCMC inference algorithm that uses a custom schedule of MCMC moves that includes proposals to change sub-trees of the PCFG parse tree. Turing does not support this algorithm. We also implemented a variant of the algorithm in Gen that uses the *recurse* combinator to cache intermediate computations when evaluating the covariance function. The results in Table 2 show that Gen gives a >10x speedup over Venture, even without caching, at a cost of 1.7x greater code size. The Gen implementation without caching gives comparable performance to the Julia implementation, with a >4x reduction in code size. Finally, *recurse* gives a 1.7x speedup at the cost of somewhat more code. These results indicate that Gen substantially reduces code size while remaining competitive with a raw Julia implementation.

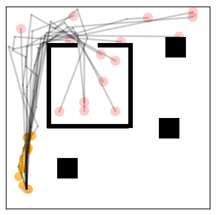


(a) Dataset and Predictions **(b)** Accuracy vs. Runtime

Figure 7. Predictions and accuracy for GP structure learning.

Table 2. Evaluation results for GP structure learning.

	Runtime (ms/step)	LOC (approx.)
Gen (Dynamic + *recurse*)	4.96ms (± 0.15)	230
Julia (handcoded, no caching)	6.17ms (± 0.45)	440
Gen (Dynamic)	8.37ms (± 0.53)	100
Venture (no caching)	107.01ms (± 6.57)	60



	Runtime (sec.)
Gen (Static)	0.77s (± 0.12)
Turing	2.57s (± 0.04)
Gen (Dynamic)	3.14s (± 0.22)

Figure 8. Left: Observed locations of agent (orange), inferred path (gray), and destination (red). Black polygons are obstacles. Right: Evaluation results for 1000 iterations of MCMC.

6.3 Algorithmic Model of an Autonomous Agent

We next consider inference in model of an autonomous agent that uses a rapidly exploring random tree [25] path planning algorithm to model the agent's goal-directed motion [8]. The task is to infer the destination of the agent from observations of its location over time. We evaluated two Gen implementations and one Turing implementation of the same MCMC inference algorithm. Gen and Turing, unlike Venture, use probabilistic DSLs that are embedded in a high-performance host language, which permits seamless application of MCMC to models that use fast simulations like path planners. The results in Figure 8 show that the Gen Static DSL outperformed the Dynamic DSL by roughly 4x, and Turing by 3.3x.

6.4 Nonlinear State-Space Model

We next consider marginal likelihood estimation in a nonlinear state-space model. We tested two particle filtering inference algorithms [9] implemented in Gen, one using a default 'prior' proposal distribution and one using a custom 'optimal' proposal derived by manual analysis of the model. Turing does not support using custom proposals with particle filtering. We implemented only the default proposal variant in Turing. The results in Figure 9 show that the custom proposal gives accurate results in an order of magnitude less time than the default proposal. Also, Table 3 shows that the Gen Static DSL implementation using the default proposal outperforms the Turing implementation of the same algorithm by a factor of 23x. Finally, we see that the *unfold* combinator provides a speedup of 12x for the Dynamic DSL.

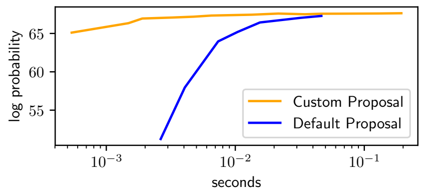


Figure 9. Comparing default and custom proposals for particle filtering. Accuracy is the log marginal likelihood estimate.

Table 3. Evaluation results for state-space model.

	Runtime (ms)	LOC
Gen (Static + *unfold* + default proposal)	13ms (± 2)	33
Gen (Dynamic + *unfold* + default proposal)	78ms (± 7)	33
Turing (default proposal)	306ms (± 153)	20
Gen (Dynamic + default proposal)	926ms (± 66)	26

6.5 3D Body Pose Estimation from Depth Images

We next consider an inference task from computer vision [24, 52]. The model shown in Figure 6 posits a 3D articulated mesh model of a human body, parametrized by pose variables including joint rotations, and a rendering process by which the mesh projects onto a two-dimensional depth image. The task is to infer the underlying 3D pose parameters from a noisy depth image. We implemented two importance sampling algorithms, one with a default proposal and one with a custom proposal that employs a deep neural network trained on data generated from the model. Neither Turing nor Venture support custom importance sampling proposals. We compared a Turing implementation using the default proposal, with a Gen implementation using the custom proposal. The Gen proposal is a generative function written in the Static DSL that invokes a generative function written in the TensorFlow DSL. The results (Figure 6) show that the Gen implementation gives more accurate results in orders of magnitude less time than Turing. This shows that custom proposals trained using amortized inference can be essential for good performance in computer vision applications.

7 Related Work

We discuss work in probabilistic programming, differentiable programming/deep learning, and probabilistic inference.

Probabilistic Programming Researchers have introduced many probabilistic programming languages over the last 20 years [6, 13–15, 28, 30, 31, 35, 36, 41, 44, 48]. With the exception of Venture and Turing, these languages do not support user-programmable inference [29]. Users of Venture and Turing specify inference algorithms in restrictive inference DSLs. In contrast, Gen users define inference algorithms in ordinary Julia code that manipulates execution traces of probabilistic models. The examples in this paper show how Gen's approach to programmable inference is more flexible and supports modular and reusable custom inference code.

Compilers have been developed for probabilistic programming languages to improve the performance of inference [6, 21, 50, 51]. Gen's JIT compiler is currently less aggressive than these compilers, but Gen's architecture is significantly more extensible, allowing end users to integrate custom proposal distributions, new inference algorithms, custom combinators that influence the control flow of modeling and inference, and custom compilers for new DSLs. No existing system with compiled inference supports these features.

The TensorFlow DSL in Gen provides high-performance support for deep generative models, including those written in the style favored by Pyro [5] and Edward [44]. However, unlike Pyro, Gen can also express efficient Markov chain and sequential Monte Carlo algorithms that rely on efficient mutation of large numbers of traces. Also, these systems lack Gen's programmable inference support and hierarchical address spaces.

Deep Learning and Differentiable Programming Unlike deep learning platforms such as TensorFlow [1], PyTorch [34], Theano [3], and MXNet [7], Gen programs explicitly factorize modeling and inference. Gen is also more expressive. For example, Gen provides support for fast Monte Carlo and numerical optimization updates to model state. Gen allows for models and inference algorithms to be specified in flexible combinations of Julia, TensorFlow, and other DSLs added by Gen users. Gen also automates the process of calculating the proposal densities needed for a broad range of advanced Monte Carlo techniques, given user-specified custom proposals that can combine Julia and TensorFlow code.

Probabilistic Inference Gen provides language constructs, DSLs, and inference library routines for state-of-the-art techniques from probabilistic modeling and inference, including generative modeling [32], deep neural networks [1], Monte Carlo inference [2, 16, 38], and numerical optimization [17]. Recently, artificial intelligence and machine learning researchers have explored combinations of these techniques. For example, recent work in computer vision uses generative models based on graphics pipelines and performs inference by combining Monte Carlo, optimization, and machine learning [24, 49]. Gen's language constructs support these kinds of sophisticated hybrid architectures. This is illustrated via the case studies in this paper, which show how Gen can implement state-of-the-art algorithms for (i) inferring 3D body pose by inverting a generative model based on a graphics engine; (ii) inferring the probable goals of an agent by inverting an algorithmic planner [8]; and (iii) learning the structure of Gaussian process covariance functions for modeling time series data [11, 29, 40, 42].

8 Conclusion

This paper has shown how to build a probabilistic programming system that solves challenging problems from multiple fields, with performance that is superior to other state-of-the-art probabilistic programming systems. Key challenges that were overcome include (i) achieving good performance for heterogeneous probabilistic models that combine different types of computations such as simulators, deep neural networks, and recursion; and (ii) providing users with abstractions that simplify the implementation of custom inference algorithms while being minimally restrictive.

Acknowledgments

This research was supported in part by the US Department of the Air Force contract FA8750-17-C-0239, grants from the MIT Media Lab/Harvard Berkman Center Ethics & Governance of AI Fund and the MIT CSAIL Systems that Learn Consortium, a gift from the Aphorism Foundation, and the US Department of Defense through the the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

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] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. 2010. Particle Markov chain Monte Carlo Methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72, 3 (2010), 269–342.

[3] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian J. Goodfellow, Arnaud Bergeron, and Yoshua Bengio. 2011. Theano: Deep learning on GPUs with Python. In *Big Learn Workshop, NIPS 2011*.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 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] 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.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. (2015). arXiv:1512.01274

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

[9] 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.

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

[11] 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.

[12] Martin A. Fischler and Robert C. Bolles. 1981. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* 24, 6 (June 1981), 381–395.

[13] 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.

[14] 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.

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

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

[17] Andreas Griewank. 1989. On automatic differentiation. *Mathematical Programming: Recent Developments and Applications* 6, 6 (1989), 83–107.

[18] 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.

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

[20] Geoffrey E Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. 1995. The" wake-sleep" algorithm for unsupervised neural networks. *Science* 268, 5214 (1995), 1158–1161.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. 2017. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*. PMLR, 1682–1690.

[27] Jonathan Malmaud and Lyndon White. 2018. TensorFlow.jl: An idiomatic Julia front end for TensorFlow. *Journal of Open Source Software* 3, 31 (2018), 1002.

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

[29] 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.

[30] 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)*, Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. Williams, and Aron Culotta (Eds.). Curran Associates, 1249–1257.

[31] 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)*. Morgan Kaufmann Publishers Inc., 1352–1359.

[32] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective.*

[33] Lawrence M. Murray. 2013. Bayesian state-space modelling on high-performance hardware using LibBi. (2013). arXiv:1306.3277

[34] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

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

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

[37] Christian Robert and George Casella. 2013. *Monte Carlo Statistical Methods.* Springer.

[38] Gareth O. Roberts and Jeffrey S. Rosenthal. 1998. Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 60, 1 (1998), 255–268.

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

[40] 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, Article 37 (2019), 29 pages. (Forthcoming, 46th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'19).

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

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

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

[44] 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)*.

[45] Chong Wang and David M. Blei. 2018. A general method for robust Bayesian modeling. *Bayesian Analysis* 13, 4 (12 2018), 1163–1191.

[46] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. 2009. Password Cracking Using Probabilistic Context-Free Grammars. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 391–405.

[47] 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.

[48] 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.

[49] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. 2017. Neural Scene De-rendering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017)*. IEEE, 7035–7043.

[50] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled inference for probabilistic programming languages. (2016). arXiv:1606.09242

[51] 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.

[52] 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.