Compiling Stan to Generative Probabilistic Languages and Extension to Deep Probabilistic Programming

Guillaume Baudart
INRIA Paris
École normale supérieure – PSL University
France

Javier Burroni
UMass Amherst
USA

Martin Hirzel
MIT-IBM Watson AI Lab, IBM Research
USA

Louis Mandel
MIT-IBM Watson AI Lab, IBM Research
USA

Avraham Shinnar
MIT-IBM Watson AI Lab, IBM Research
USA

Abstract

Stan is a probabilistic programming language that is popular in the statistics community, with a high-level syntax for expressing probabilistic models. Stan differs by nature from generative probabilistic programming languages like Church, Anglican, or Pyro. This paper presents a comprehensive compilation scheme to compile any Stan model to a generative language and proves its correctness. We use our compilation scheme to build two new backends for the Stanc3 compiler targeting Pyro and NumPyro. Experimental results show that the NumPyro backend yields a 2.3x speedup compared to Stan in geometric mean over 26 benchmarks. Building on Pyro we extend Stan with support for explicit variational inference guides and deep probabilistic models. That way, users familiar with Stan get access to new features without having to learn a fundamentally new language.

CCS Concepts: • **Software and its engineering → Compilers**; • **Theory of computation → Probabilistic computation**.

Keywords: Probabilistic programming, Semantics, Stan, Pyro

ACM Reference Format:
Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. 2021. Compiling Stan to Generative Probabilistic Languages and Extension to Deep Probabilistic Programming. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454058

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454058

1 Introduction

Probabilistic Programming Languages (PPLs) are designed to describe probabilistic models and run inference on them. There exists a variety of PPLs. BUGS [21], JAGS [26], and Stan [7] focus on efficiency, constraining what is expressible to a subset of models which support fast inference techniques. These languages enjoy broad adoption by the statistics and social sciences communities [6, 10, 11]. *Generative languages* like Church [12], Anglican [32], WebPPL [13], Pyro [3], and Gen [9] describe *generative models*, i.e., stochastic procedures that simulate the data generation process. Coming from a core programming languages heritage, generative PPLs typically support rich control constructs and models over structured data. Generative PPLs are increasingly used in machine-learning research and are rapidly incorporating new ideas, such as Stochastic Gradient Variational Inference (SVI), in what is now called Deep Probabilistic Programming [2, 3, 33].

While the semantics of probabilistic languages have been extensively studied [14, 15, 18, 30], to the best of our knowledge little is known about the relationship between Stan and generative PPLs. We show that a simple 1:1 translation is incorrect or incomplete for a set of subtle but widely-used Stan features, such as left expressions or implicit priors.

This paper formalizes the relationship between Stan and generative PPLs and introduces, with correctness proof, a *comprehensive* compilation scheme that can compile any Stan program to a generative PPL. This enables leveraging the rich set of existing Stan models for testing, benchmarking, or experimenting with new features or inference techniques. Based on this compilation scheme we implemented two new backends for the Stanc3 compiler targeting Pyro [3] and NumPyro [25], a JAX [5] based version of Pyro. Both Pyro and NumPyro runtimes offer NUTS [16] (No U-Turn Sampler), an optimized Hamiltonian Monte-Carlo (HMC) algorithm that is the preferred inference method for Stan. We can thus validate our approach against Stan. Results show that models compiled using our NumPyro backend yield equivalent results while being 2.3x faster than

their Stan counterpart in the geometric mean over 26 benchmarks. Our compiler and runtime library are open-source at https://github.com/deepppl.

In addition, recent probabilistic languages offer new features to program and reason about complex models. Our compilation scheme combined with conservative extensions of Stan can be used to make these benefits available to Stan users. As a proof of concept, based on our Pyro backend, this paper introduces DeepStan: Stan extended with support for explicit variational guides and deep probabilistic models. Variational inference was central in the design of Pyro and programmers can easily craft their own inference guides to run variational inference on probabilistic models. Pyro is built on top of PyTorch [24]. Programmers can thus seamlessly import neural networks designed with the state-of-the-art API provided by PyTorch.

This paper makes the following contributions:

- A comprehensive compilation scheme from Stan to a generative PPL (Section 2).
- Correctness proof of the compilation scheme (Section 3).
- An open-source implementation of two new backends for Stanc3 targeting Pyro and NumPyro (Section 4).
- An extension of Stan with explicit variational inference guides and deep probabilistic models (Section 5).

The fundamental new result of this paper is that every Stan program can be expressed as a generative probabilistic program. Besides advancing the understanding of probabilistic programming languages at a fundamental level, this paper aims to provide practical benefits to the communities of both Stan and Pyro. From the perspective of the Stan community, this paper presents a new competitive compiler backend and additional capabilities while retaining familiar syntax and semantics. This compiler can thus be used to migrate existing Stan codebases to Pyro and NumPyro. From the perspective of the Pyro community, this paper presents a new compiler frontend that unlocks many existing real-world models as examples and benchmarks.

An extended version of the paper with appendices presenting the proofs and the evaluation results is available [1].

2 Overview

This section shows how to compile Stan [7], which specifies a joint probability distribution, to a generative PPL like Church, Anglican, or Pyro. This translation also demonstrates that Stan's expressive power is at most as large as that of generative languages, a fact that was not clear before our paper.

As a running example, consider the biased coin model in Figure 1. Stan's `data` block defines observed variables for N coin flips x_i, $i \in [1:N]$, which can be 0 for tails or 1 for heads. The `parameters` block introduces a latent variable $z \in [0,1]$ for the bias of the coin. The `model` block sets the prior of the bias z to Beta(1, 1), i.e., a uniform distribution

```
data {
  int N;
  int<lower=0,upper=1> x[N]; }
parameters {
  real<lower=0,upper=1> z; }
model {
  z ~ beta(1, 1);
  for (i in 1:N) x[i] ~ bernoulli(z); }
```

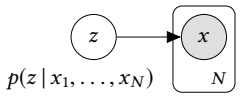


Figure 1. Biased coin model in Stan.

```
def model(N, x):
  z = sample(
        beta(1.,1.))
  for i in range(0, N):
    observe(
      bernoulli(z), x[i])
  return z
```

(a) Generative scheme

```
def model(N, x):
  z = sample(uniform(0.,1.))
  observe(beta(1.,1.), z)
  for i in range(0, N):
    observe(
      bernoulli(z), x[i])
  return z
```

(b) Comprehensive scheme

Figure 2. Compiled coin model of Figure 1.

over $[0,1]$. The `for` loop indicates that coin flips x_i are independent and identically distributed (IID) and depend on z via a Bernoulli distribution. Given concrete observed coin flips, inference yields a posterior distribution for z conditioned on $x_1, \ldots, x_N$.

2.1 Generative Translation

Generative PPLs are general-purpose languages extended with two probabilistic constructs [14, 30, 34]: `sample(`D`)` generates a sample from a distribution D and `factor(`v`)` assigns a score v to the current execution trace. Typically, `factor` is used to condition the model on input data [32]. We also introduce `observe(`D`,`v`)` as a syntactic shortcut for `factor(`D_{pdf}`(`v`))` where D_{pdf} denotes the probability density function of D. This construct penalizes executions according to the score of v w.r.t. D which captures the assumption that the observed data v follows the distribution D.

Compilation. Stan uses the same syntax `v ~ `D for both observed and latent variables. The distinction comes from the kind of the left-hand-side variable: observed variables are declared in the `data` block, latent variables are declared in the `parameters` block. A straightforward *generative translation* compiles a statement `v ~ `D into `v = sample(`D`)` if `v` is a parameter or `observe(`D`, v)` if `v` is data. For example, Figure 2a shows the compiled (using the generative scheme) version of the Stan model of Figure 1 in Python syntax.

2.2 Non-generative Features

In Stan, a model represents the unnormalized density of the joint distribution of the parameters defined in the `parameters` block given the data defined in the `data` block [7, 15]. A Stan program can thus be viewed as a function from parameters

Table 1. Stan features that defy generative translation: prevalence, example, and compilation.

Feature	%	Example	Compilation
Left expression	15	`sum(phi) ~ normal(0, 0.001*N);`	`observe(Normal(0.,0.001*N), sum(phi))`
Multiple updates	8	`phi_y ~ normal(0,sigma_py);` `phi_y ~ normal(0,sigma_pt)`	`observe(Normal(0.,sigma_py), phi_y);` `observe(Normal(0.,sigma_pt), phi_y)`
Implicit prior	58	`real alpha0;` `/* missing 'alpha0 ~ ...' */`	`alpha0 = sample(improper_uniform())`

and data to the value of a special variable `target` that represents the log-density of the model. A Stan model can be described using classic imperative statements, plus two special statements that modify the value of `target`. The first one, `target+=` e, increments the value of `target` by e. The second one, $e \sim D$, is equivalent to `target+=` $D_{\text{lpdf}}(e)$ [15] where D_{lpdf} denotes the log probability density function of D.

Unfortunately, these constructs allow the definition of models that cannot be translated using the generative translation defined above. Table 1 lists the Stan features that are not handled correctly. A *left expression* is where the left-hand-side of ~ is an arbitrary expression. *Multiple updates* occur when the same parameter appears on the left-hand-side of multiple ~ statements. An *implicit prior* occurs when there is no explicit ~ statement in the model for a parameter.

The "%" column of Table 1 indicates the percentage of Stan models that exercise each of the non-generative features among the 531 valid files in https://github.com/stan-dev/example-models. The example column contains illustrative excerpts from such models. Since these are official and long-standing examples, we assume that they use the non-generative features on purpose. Comments in the source code further corroborate that the programmer knowingly used the features. While some features only occur in a minority of models, their prevalence is too high to ignore.

2.3 Comprehensive Translation

The previous section illustrates that Stan is centered around the definition of `target`, not around generating samples for parameters, which is required by generative PPLs. The comprehensive translation adds an initialization step to generate samples for all the parameters and compiles all Stan ~ statements as observations. Parameter initialization draws from the uniform distribution in their definition domain. For the biased coin example, the result of this translation is shown in Figure 2b: The parameter `z` is first sampled uniformly on its definition domain and then conditioned with an observation.

The compilation column of Table 1 illustrates the translation of non-generative features. Left expression and multiple updates are simply compiled into observations. Parameter initialization uses the uniform distribution over its definition domain. For unbounded domains, we introduce new distributions (e.g., `improper_uniform`) with a constant density that can be normalized away during inference. Section 3.3 details the complete compilation scheme.

Intuition of correctness. The semantics of Stan as described in [15] is a classic imperative semantics. Its environment includes the special variable `target`, the unnormalized log-density of the model. On the other hand, the semantics of a generative PPL as described in [30] defines a kernel mapping an environment to a measurable function. Our compilation scheme adds uniform initializations for all parameters which comes down to the Lebesgue measure on the parameters space, and translates all ~ statements to `observe` statements. We can then show that a succession of `observe` statements yields a distribution with the same log-density as the Stan semantics. Section 3.4 details the correctness proof.

Implementation. The comprehensive compilation scheme can compile any Stan program to a generative PPL. Section 4 discusses the implementation of two new backends for the Stanc3 compiler targeting Pyro [3] – a PPL in the line of WebPPL [13] – and NumPyro – a JAX [5] based version of Pyro. Section 6 experimentally validates that our backends can compile most existing Stan models. Results also show that models compiled using our NumPyro backend outperform their Stan counterpart on existing benchmarks.

Extensions. Pyro is a *deep universal probabilistic programming languages* with native support for variational inference. Building on Pyro, we use our compiler to extend Stan with support for explicit variational guides (Section 5.1) and deep neural networks to capture complex relations between parameters (Section 5.2).

3 Semantics and Compilation

This section, building on previous work, first formally defines the semantics of Stan (Section 3.1) and the semantics of GProb, a small generative probabilistic language (Section 3.2). It then defines the compilation function from Stan to GProb (Section 3.3) and proves its correctness (Section 3.4).

3.1 Stan: A Declarative Probabilistic Language

The Stan language is informally described in [7]. A Stan program is a sequence of blocks which in order: declares functions, declares input names and types, pre-processes input

data, declares the parameters to infer, defines transformations on parameters, defines the model, and post-processes the parameters. The only mandatory block is `model`. Variables declared in a block are visible in subsequent blocks.

$$
\begin{array}{rcl}
\textit{program} & ::= & \texttt{functions}\ \{\textit{fundecl}^*\}\ ? \\
& & \texttt{data}\ \{\textit{decl}^*\}\ ? \\
& & \texttt{transformed data}\ \{\textit{decl}^*\ \textit{stmt}\}\ ? \\
& & \texttt{parameters}\ \{\textit{decl}^*\}\ ? \\
& & \texttt{transformed parameters}\ \{\textit{decl}^*\ \textit{stmt}\}\ ? \\
& & \texttt{model}\ \{\textit{decl}^*\ \textit{stmt}\} \\
& & \texttt{generated quantities}\ \{\textit{decl}^*\ \textit{stmt}\}\ ?
\end{array}
$$

Variable declarations ($\textit{decl}^*$) are lists of variables names with their types (e.g., `int N;`) or arrays with their type and shape (e.g., `real x[N]`). Types can be constrained to specify the domain of a variable (e.g., `real <lower=0> x` for $x \in \mathbb{R}^+$). Note that `vector` and `matrix` are primitive types that can be used in arrays (e.g. `vector[N] x[10]` is an array of `10` vectors of size `N`). Shapes and sizes of arrays, matrices, and vectors are explicit and can be arbitrary expressions.

$$
\begin{array}{rcl}
\textit{decl} & ::= & \textit{base\_type constraint}\ x\ \texttt{;} \\
& \mid & \textit{base\_type constraint}\ x\ \texttt{[}\textit{shape}\texttt{]}\ \texttt{;} \\
\textit{base\_type} & ::= & \texttt{real} \mid \texttt{int} \\
& \mid & \texttt{vector[}\textit{size}\texttt{]} \mid \texttt{matrix[}\textit{size}\texttt{,}\textit{size}\texttt{]} \\
\textit{constraint} & ::= & \varepsilon \mid \texttt{<lower =}\ e\ \texttt{, upper =}\ e\ \texttt{>} \\
& \mid & \texttt{<lower =}\ e\ \texttt{>} \mid \texttt{<upper =}\ e\ \texttt{>} \\
\textit{shape} & ::= & \textit{size} \mid \textit{shape}\ \texttt{,}\ \textit{size} \\
\textit{size} & ::= & e
\end{array}
$$

Inside a block, Stan is similar to a classic imperative language, with two extra, specialized statements: `target += e` directly updates the log-density of the model (stored in the reserved variable `target`), and $x \sim D$ indicates that a variable x follows a distribution D.

$\textit{stmt}$ ::=	$x = e$	variable assignment
	$x[e_1, \ldots, e_n] = e$	array assignment
	$\textit{stmt}_1\ \texttt{;}\ \textit{stmt}_2$	sequence
	$\texttt{for (}x\ \texttt{in}\ e_1\texttt{:}e_2\texttt{) \{}\textit{stmt}\texttt{\}}$	loop over a range
	$\texttt{for (}x\ \texttt{in}\ e\texttt{) \{}\textit{stmt}\texttt{\}}$	loop over a collection
	$\texttt{while (}e\texttt{) \{}\textit{stmt}\texttt{\}}$	while loop
	$\texttt{if (}e\texttt{)}\ \textit{stmt}_1\ \texttt{else}\ \textit{stmt}_2$	conditional
	`skip`	no operation
	`target +=` e	direct log-density update
	$e \sim f(e_1, \ldots, e_n)$	probability distribution

Expressions comprise constants, variables, arrays, vectors, matrices, access to elements of an indexed structure, and function calls (also used to model binary operators):

$$
\begin{array}{l}
e ::= c \mid x \mid f(e_1, \ldots, e_n) \mid \{e_1, \ldots, e_n\} \mid [e_1, \ldots, e_n] \\
\quad \mid [[e_{1_1}, \ldots, e_{1_m}], \ldots, [e_{n_1}, \ldots, e_{n_m}]] \mid e_1[e_2]
\end{array}
$$

Semantics. Stan programs are evaluated in three steps:

1. data preparation with `data` and `transformed data`
2. inference over the model defined by `parameters`, `transformed parameters`, and `model`
3. post-processing with `generated quantities`.

Sections `transformed data`, `transformed parameters`, and `generated quantities` introduce new variables. Semantically, these sections can all be inlined in the `model` section. Any Stan program can thus be rewritten into an equivalent program with only the three blocks `data`, `parameters`, and `model`. Alternatively, we show in Section 3.3 that the `transformed data` section can be pre-computed and passed as input to the model, and the `generated quantities` can be post-processed after the inference.

$$
\begin{array}{l}
\texttt{functions}\ \{\textit{fundecls}\} \\
\texttt{data}\ \{\textit{decls}_d\} \\
\texttt{transformed data} \\
\{\textit{decls}_{td}\ \textit{stmt}_{td}\} \\
\texttt{parameters}\ \{\textit{decls}_p\} \\
\texttt{transformed parameters} \\
\qquad \{\textit{decls}_{tp}\ \textit{stmt}_{tp}\} \\
\texttt{model}\ \{\textit{decls}_m\ \textit{stmt}_m\} \\
\texttt{generated quantities} \\
\qquad \{\textit{decls}_g\ \textit{stmt}_g\}
\end{array}
\quad \equiv \quad
\begin{array}{l}
\texttt{data}\ \{\textit{decls}_d\} \\
\texttt{parameters}\ \{\textit{decls}_p\} \\
\texttt{model}\ \{ \\
\quad \textit{decls}_{td}\ \textit{decls}_{tp}\ \textit{decls}_m\ \textit{decls}_g \\
\quad \textit{stmt}'_{td}\ \textit{stmt}'_{tp}\ \textit{stmt}'_m\ \textit{stmt}'_g \\
\}
\end{array}
$$

Functions declared in `functions` are inlined ($\textit{stmt}'$ is equivalent to $\textit{stmt}$ after inlining). To simplify the presentation, we focus on this simplified language.

Notations. To refer to the different parts of a program, we will use the following functions. For a Stan program $p = \texttt{data}\ \{\textit{decls}_d\}\ \texttt{parameters}\ \{\textit{decls}_p\}\ \texttt{model}\ \{\textit{decls}_m\ \textit{stmt}\}$:

$$\textit{data}(p) = \textit{decls}_d \quad \textit{params}(p) = \textit{decls}_p \quad \textit{model}(p) = \textit{stmt}$$

In the following, an environment $\gamma : \textit{Var} \to \textit{Val}$ is a mapping from variables to values, $\gamma(x)$ returns the value of the variable x in an environment γ, $\gamma[x \leftarrow v]$ returns the environment γ where the value of x is set to v, and γ_1, γ_2 denotes the union of two environments.

The notation $\int_X \mu(dx) f(x)$ is the integral of f w.r.t. the measure μ where $x \in X$ is the integration variable. When μ is the Lebesgue measure we also write $\int_X f(x)dx$.

Following [15], we define the semantics of the model block as a deterministic function that takes an initial environment containing the input data and the parameters, and returns an updated environment where the value of the variable `target` is the un-normalized log-density of the model.

We can then define the semantics of a Stan program as a *kernel* [18, 30, 31], that is, a function $\{\!|p|\!\} : \mathcal{D} \to \Sigma_X \to [0, \infty]$ where Σ_X denotes the σ-algebra of the parameter domain X, that is, the set of measurable sets of the product space of parameter values. Given an environment D containing the input data, $[\![p]\!]_D$ is a *measure* that maps a measurable set of parameter values U to a score in $[0, \infty]$ obtained by integrating the density of the model, $\exp(\texttt{target})$, over all the possible parameter values in U.

$$\{\!|p|\!\}_D = \lambda U. \int_U \exp([\![\textit{model}(p)]\!]_{D[\textit{params}(p) \leftarrow \theta]}(\texttt{target}))\, d\theta$$

$$
\begin{aligned}
[\![x = e]\!]_\gamma &= \gamma[x \leftarrow [\![e]\!]_\gamma] \\
[\![x[e_1,\ldots,e_n] = e]\!]_\gamma &= \gamma[x \leftarrow (x[[\![e_1]\!]_\gamma,\ldots,[\![e_n]\!]_\gamma] \leftarrow [\![e]\!]_\gamma)] \\
[\![s_1\,;\, s_2]\!]_\gamma &= [\![s_2]\!]_{[\![s_1]\!]_\gamma} \\
[\![\texttt{for}\ (x\ \texttt{in}\ e_1 : e_2)\ \{s\}]\!]_\gamma &= \\
&\quad \mathit{let}\ n_1 = [\![e_1]\!]_\gamma\ \mathit{in}\ \mathit{let}\ n_2 = [\![e_2]\!]_\gamma\ \mathit{in} \\
&\quad \mathit{if}\ n_1 > n_2\ \mathit{then}\ \gamma\ \mathit{else}\ [\![\texttt{for}\ (x\ \texttt{in}\ n_1 + 1 : n_2)\ \{s\}]\!]_{[\![s]\!]_{\gamma[x \leftarrow n_1]}} \\
[\![\texttt{while}\ (e)\ \{s\}]\!]_\gamma &= \mathit{if}\ [\![e]\!]_\gamma = 0\ \mathit{then}\ \gamma\ \mathit{else}\ [\![\texttt{while}\ (e)\ \{s\}]\!]_{[\![s]\!]_\gamma} \\
[\![\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2]\!]_\gamma &= \mathit{if}\ [\![e_1]\!]_\gamma \neq 0\ \mathit{then}\ [\![s_1]\!]_\gamma\ \mathit{else}\ [\![s_2]\!]_\gamma \\
[\![\texttt{skip}]\!]_\gamma &= \gamma \\
[\![\texttt{target}\ \texttt{+=}\ e]\!]_\gamma &= \gamma[\texttt{target} \leftarrow \gamma(\texttt{target}) + [\![e]\!]_\gamma] \\
[\![e_1\ \texttt{\~{}}\ e_2]\!]_\gamma &= \mathit{let}\ D = [\![e_2]\!]_\gamma\ \mathit{in}\ [\![\texttt{target}\ \texttt{+=}\ D_{lpdf}(e_1)]\!]_\gamma
\end{aligned}
$$

Figure 3. Semantics of statements

Given the input data, the posterior distribution of a Stan program is obtained by normalizing the measure $\{\!|p|\!\}_D$. As the integrals are often intractable, the runtime uses approximate inference schemes to compute the posterior distribution.

We now detail the semantics of statements and expressions in a model block. This formalization is similar to the semantics proposed in [15] but expressed denotationally.

Statements. The semantics of a statement $[\![s]\!] : (\mathit{Var} \to \mathit{Val}) \to (\mathit{Var} \to \mathit{Val})$ is a function from an environment γ to an updated environment. Figure 3 gives the semantics of Stan statements. The initial environment contains the input data, the parameters, and the reserved variable `target` initialized to `0`. An assignment updates the value of a variable or of a cell of an indexed structure in the environment. A sequence $s_1\,;\, s_2$ evaluates s_2 in the environment produced by s_1. A `for` loop on ranges first evaluates the value of the bounds n_1 and n_2 and then repeats the execution of the body $1 + n_2 - n_1$ times. Iterations over indexed structures (`for (x in e) {s}`) are syntactic sugar over loops on ranges. The behavior depends on the underlying type. For vectors and arrays, iteration is limited to one dimension.

$$
\begin{aligned}
[\![\texttt{for}\ (x\ \texttt{in}\ e)\ \{s\}]\!]_\gamma = {} & \mathit{let}\ v = [\![e]\!]_\gamma\ \mathit{in} \qquad (i \text{ is a fresh variable}) \\
& [\![\texttt{for}\ (i\ \texttt{in}\ 1 : \mathit{length}(v))\ \{x = v[i]\,;\, s\}]\!]_\gamma
\end{aligned}
$$

For matrices, iteration is over the two dimensions:

$$
\begin{aligned}
&[\![\texttt{for}\ (x\ \texttt{in}\ e)\ \{s\}]\!]_\gamma = \qquad (i \text{ and } j \text{ are fresh variables}) \\
&\quad \mathit{let}\ v = [\![e]\!]_\gamma\ \mathit{in} \\
&\quad \left[\!\!\left[\begin{array}{l}\texttt{for}\ (i\ \texttt{in}\ 1 : \mathit{length}(v)) \\ \quad \texttt{for}\ (j\ \texttt{in}\ 1 : \mathit{length}(v[i]))\ \{x = v[i][j]\,;\, s\}\end{array}\right]\!\!\right]_\gamma
\end{aligned}
$$

A `while` loop repeats the execution of its body while the condition is not 0. An `if` statement executes one of the two branches depending on the value of the condition. A `skip` leaves the environment unchanged. A statement `target +=` e adds the value of e to `target` in the environment. Finally, a

$$
\begin{aligned}
[\![c]\!]_\gamma &= c & [\![\{e_1,\ldots,e_n\}]\!]_\gamma &= \{[\![e_1]\!]_\gamma,\ldots,[\![e_n]\!]_\gamma\} \\
[\![x]\!]_\gamma &= \gamma(x) & [\![[e_1,\ldots,e_n]]\!]_\gamma &= [[\![e_1]\!]_\gamma,\ldots,[\![e_n]\!]_\gamma] \\
[\![e_1[e_2]]\!]_\gamma &= [\![e_1]\!]_\gamma[[\![e_2]\!]_\gamma] & [\![f(e)]\!]_\gamma &= f([\![e]\!]_\gamma)
\end{aligned}
$$

Figure 4. Semantics of expressions

statement e_1 `~` e_2 evaluates the expression e_2 into a probability distribution D and updates the target with the value of the log-density of D at e_1.

Expressions. The semantics of an expression $[\![e]\!] : (\mathit{Var} \to \mathit{Val}) \to \mathit{Val}$ is a function from a environment to values. Figure 4 gives the semantics of Stan expressions. Constants evaluate to themselves. Variables are looked up in the environment. Arrays, vectors, and matrix expressions evaluate all their components. Indexing expressions obtain the corresponding value in the associated data. Function calls apply the function to the value of the arguments. Functions are built-ins like + or `normal` (user-defined functions are inlined).

Limitations. We consider only terminating programs which means in particular that all loops perform a bounded number of iterations. We also limit the access and update of `target` to the statements `target +=` e and e_1 `~` e_2.

Assumption 1. *All programs terminate.*

Assumption 2. *Expressions cannot depend on* `target`.

3.2 GProb: A Simple Generative PPL

To formalize the compilation, we first define the target language: GProb, a simple generative probabilistic language similar to the one defined in [30]. GProb is an expression language with the following syntax:

$$
\begin{aligned}
e \ ::= \ & c \mid x \mid \{e_1,\ldots,e_n\} \mid [e_1,\ldots,e_n] \mid e_1[e_2] \mid f(e_1,\ldots,e_n) \\
\mid \ & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ x[e_1,\ldots,e_n] = e\ \texttt{in}\ e' \\
\mid \ & \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid \texttt{for}_{\mathcal{X}}\ (x\ \texttt{in}\ e_1 : e_2)\ e_3 \mid \texttt{while}_{\mathcal{X}}\ (e_1)\ e_2 \\
\mid \ & \texttt{factor}(e) \mid \texttt{sample}(e) \mid \texttt{return}(e)
\end{aligned}
$$

An expression is either a Stan expression, a local binding (`let`), a conditional (`if`), or a loop (`for` or `while`). To simplify the presentation, loops are parameterized by the set $\mathcal{X}$ of variables updated and returned by their body. Moreover, we limit the definition of the semantics to terminating loops that do not depend on sampled values in the body of the loop. GProb also contains the classic probabilistic expressions: `sample` draws a sample from a distribution, and `factor` assigns a score to the current execution trace to condition the model. The `return` expression lifts a deterministic expression to a probabilistic context.

We also introduce `observe`(D, v) as a syntactic shortcut for `factor`$(D_{\text{pdf}}(\texttt{v}))$ where D_{pdf} denotes the density function of D. This construct penalizes the current execution with the likelihood of v w.r.t. D which captures the assumption that the observed data v follows the distribution D.

$$
\begin{aligned}
&\{\!|\texttt{return}(e)|\!\}_\gamma &&= \lambda U.\, \delta_{[\![e]\!]_\gamma}(U)\\
&\{\!|\texttt{let}\ x = e_1\ \texttt{in}\ e_2|\!\}_\gamma &&= \lambda U.\, \int_X \{\!|e_1|\!\}_\gamma(dv) \times \{\!|e_2|\!\}_{\gamma[x\leftarrow v]}(U)\\
&\{\!|\texttt{let}\ x\texttt{[}e_1,\ldots,e_n\texttt{]} = e\ \texttt{in}\ e'|\!\}_\gamma = \\
&\quad \lambda U.\, \int_X \{\!|e|\!\}_\gamma(dv) \times \{\!|e'|\!\}_{\gamma[x\leftarrow(x\texttt{[}[\![e_1]\!]_\gamma,\ldots,[\![e_n]\!]_\gamma\texttt{]}\leftarrow v)]}(U)\\
&\{\!|\texttt{for}_\mathcal{X}\ (x\ \texttt{in}\ e_1\texttt{:}e_2)\ e_3|\!\}_\gamma = \\
&\quad \lambda U.\mathit{let}\ n_1 = [\![e_1]\!]_\gamma\ \mathit{in}\ \mathit{let}\ n_2 = [\![e_2]\!]_\gamma\ \mathit{in}\\
&\qquad \mathit{if}\ n_1 > n_2\ \mathit{then}\ \delta_{\gamma(\mathcal{X})}(U)\\
&\qquad \mathit{else} \int_X \{\!|e_3|\!\}_{\gamma[x\leftarrow n_1]}(dv) \times\\
&\qquad\qquad \{\!|\texttt{for}_\mathcal{X}\ (x\ \texttt{in}\ n_1 + 1\texttt{:}n_2)\ e_3|\!\}_{\gamma[\mathcal{X}\leftarrow v]}(U)\\
&\{\!|\texttt{while}_\mathcal{X}\ (e_1)\ e_2|\!\}_\gamma\ = \\
&\quad \lambda U.\mathit{if}\ [\![e_1]\!]_\gamma = 0\ \mathit{then}\ \delta_{\gamma(\mathcal{X})}(U)\\
&\qquad \mathit{else} \int_X \{\!|e_2|\!\}_\gamma(dv) \times \{\!|\texttt{while}_\mathcal{X}\ (e_1)\ e_2|\!\}_{\gamma[\mathcal{X}\leftarrow v]}(U)\\
&\{\!|\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2|\!\}_\gamma = \lambda U.\mathit{if}\ [\![e]\!]_\gamma \neq 0\ \mathit{then}\ \{\!|e_1|\!\}_\gamma(U)\\
&\qquad\qquad\qquad\qquad \mathit{else}\ \{\!|e_2|\!\}_\gamma(U)\\
&\{\!|\texttt{sample}(e)|\!\}_\gamma &&= \lambda U.\, [\![e]\!]_\gamma\,(U)\\
&\{\!|\texttt{factor}(e)|\!\}_\gamma &&= \lambda U.\, \exp([\![e]\!]_\gamma)\delta_{()}(U)
\end{aligned}
$$

Figure 5. Generative probabilistic language semantics

Semantics. Following [30] we give a measure-based semantics to GProb. The semantics of an expression is a kernel that given an environment returns a measure on the set of possible values. Given input data, normalizing the corresponding measure computes the program's posterior distribution.

The semantics of GProb is given in Figure 5. A deterministic expression is lifted to a probabilistic expression with the Dirac delta measure ($\delta_x(U) = 1$ if $x \in U$, 0 otherwise). A local definition $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is interpreted by integrating the semantics of e_2 over the set of all possible values for x.

Compared to the language defined in [30], we added Stan-like loops. Loops behave like a sequence of expressions and return the values of the variables updated by their body. We impose that the condition of a loop cannot depend on parameters sampled in the loop body, and consider only terminating loops. Hence for any given context γ, it suffices to unroll the definition of the loop semantics a finite number of times to get a measure term describing the semantics.

Finally, the semantics of probabilistic operators is the following. The semantics of $\texttt{sample}(e)$ is the probability distribution $[\![e]\!]_\gamma$ (e.g. $\mathcal{N}(0,1)$). A type system, omitted here for conciseness, ensures that we only sample from distributions. The semantics of $\texttt{factor}(e)$ is a measure defined on the singleton space $()$ whose value is $\exp([\![e]\!])$ (this operator corresponds to `score` in [30] but in log-scale, which is common for numerical precision).

$$
\begin{aligned}
C_k\,(t\ \mathit{cstr}\ x\,;) &= \texttt{let}\ x = \texttt{sample}(C\,(\mathit{cstr}, \texttt{[]}))\ \texttt{in}\ k\\
C_k\,(t\ \mathit{cstr}\ x\texttt{[}\mathit{shape}\texttt{]};) &= \texttt{let}\ x = \texttt{sample}(C\,(\mathit{cstr}, \mathit{shape}))\ \texttt{in}\ k\\
C_k\,(\mathit{decl}\ \mathit{decls}) &= C_{C_k(\mathit{decls})}\,(\mathit{decl})\\
C\,(\varepsilon, \mathit{shape}) &= \texttt{improper\_uniform}([-\infty, \infty], \mathit{shape})\\
C\,(\texttt{<lower=}e_1\texttt{>}, \mathit{shape}) &= \texttt{improper\_uniform}([e_1, \infty], \mathit{shape})\\
C\,(\texttt{<upper=}e_2\texttt{>}, \mathit{shape}) &= \texttt{improper\_uniform}([-\infty, e_2], \mathit{shape})\\
C\,(\texttt{<lower=}e_1\texttt{,upper=}e_2\texttt{>}, \mathit{shape}) &= \texttt{uniform}([e_1, e_2], \mathit{shape})
\end{aligned}
$$

Figure 6. Comprehensive compilation of parameters

$$
\begin{aligned}
C_k\,(x = e) &= \texttt{let}\ x = \texttt{return}(e)\ \texttt{in}\ k\\
C_k\,(x\texttt{[}e_1,\ldots,e_n\texttt{]} = e) &= \texttt{let}\ x\texttt{[}e_1,\ldots,e_n\texttt{]} = e\ \texttt{in}\ k\\
C_k\,(s_1;\ s_2) &= C_{C_k(s_2)}\,(s_1)\\
C_k\,(\texttt{for}\ (x\ \texttt{in}\ e_1\texttt{:}e_2)\ \{s\}) &= \\
&\quad \texttt{let}\ \mathit{lhs}(s) = \texttt{for}_{\mathit{lhs}(s)}\ (x\ \texttt{in}\ e_1\texttt{:}e_2)\ C_{\texttt{return}(\mathit{lhs}(s))}\ (s)\ \texttt{in}\ k\\
C_k\,(\texttt{while}\ (e)\ \{s\}) &= \\
&\quad \texttt{let}\ \mathit{lhs}(s) = \texttt{while}_{\mathit{lhs}(s)}\ (e)\ C_{\texttt{return}(\mathit{lhs}(s))}\ (s)\ \texttt{in}\ k\\
C_k\,(\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2) &= \texttt{if}\ (e)\ C_k\,(s_1)\ \texttt{else}\ C_k\,(s_2)\\
C_k\,(\texttt{skip}) &= k\\
C_k\,(\texttt{target += }e) &= \texttt{let}\ () = \texttt{factor}(e)\ \texttt{in}\ k\\
C_k\,(e \sim f(e_1,\ldots,e_n)) &= \texttt{let}\ () = \texttt{observe}(f(e_1,\ldots,e_n), e)\ \texttt{in}\ k
\end{aligned}
$$

Figure 7. Comprehensive compilation of statements

3.3 Comprehensive Translation

The key idea is to first sample all parameters from priors with a constant density that can be normalized away during inference (e.g., *uniform* on bounded domains), and then compile all ~ statements into `observe` statements.

The compilation functions $C_k\,(\mathit{params}(p))$ for the parameters and $C_k\,(\mathit{model}(p))$ for the model are both parameterized by a continuation k. The compilation of the entire program first compiles the parameters to introduce the priors, then compiles the model, and finally adds a return statement for all the parameters. In continuation passing style:

$$C(p) = C_{C_{\texttt{return}(\mathit{params}(p))}(\mathit{model}(p))}\,(\mathit{params}(p))$$

Parameters. In Stan, parameters are defined on $\mathbb{R}^n$ with optional domain constraints (e.g. `<lower=0>`). For each parameter, the comprehensive translation sets the prior to either the *uniform* distribution on a bounded domain, or an improper prior with a constant density w.r.t. the Lebesgue measure that we call `improper_uniform`. The compilation

function of the parameters, defined Figure 6, thus produces a succession of sample expressions:

$$C_k\,(params(p)) \;=\; \texttt{let}\, x_1 = D_1\, \texttt{in} \ldots \texttt{let}\, x_n = D_n\, \texttt{in}\, k$$

where each D_i is either `uniform` or `improper_uniform`.

Statements. Figure 7 defines compilation for statements, $C_k\,(stmt)$, parameterized by a continuation k. Stan imperative assignments become functional updates using local bindings. Compiling the sequences chains the continuations. Updates to the target are compiled into `factor` expressions and all ~ statements are compiled into observations.

The compilation of Stan loops raises an issue. In Stan, all the variables that appear on the left-hand side of an assignment in the body of a loop are state variables that are incrementally updated at each iteration. Since GProb is a functional language, the state of the loops is made explicit. To propagate the environment updates at each iteration, loop expressions are annotated with all the variables that are assigned in their body ($lhs(stmt)$). These variables are returned at the end of the loop and can be used in the continuation.

Pre- and post-processing blocks. Section 3.1 showed that all Stan programs can be rewritten in a kernel using only the `data`, `parameters`, and `model` sections. This approach can make the model more complicated and thus, the inference more expensive. In particular, pre- and post-processing steps do not need to be computed at each inference step.

In Stan, users can define functions in the `functions` block. These functions can be compiled into functions in the target language using the comprehensive translation.

The code in the `transformed data` section only depends on variables declared in the `data` section and can be computed only once before the inference. We compile this section into a function that takes as argument the data and returns the transformed data. The variables declared in the `transformed data` section become new inputs for the model.

On the other hand, the `transformed parameters` block must be part of the model since it depends on the model parameters. This section is thus inlined in the compiled model.

Finally, the `generated quantities` block is executed only once on the inference result. It is compiled into a function of the data, transformed data, and parameters returned by the model. The `transformed parameters` block is also inlined, since generated quantities may depend on them.

3.4 Correctness of the Compilation

We can now show that a Stan program and the corresponding compiled code yield the same un-normalized measure up to a constant factor (and thus the same posterior distribution). The proof has two steps: (1) simplifying the sequence of `sample` statements introduced by the compilation of the parameters, and (2) showing that the value of the Stan `target` corresponds to the score computed by the generated code.

Priors. First, we simplify the nested integrals introduced by the sequence of `sample` statements for the parameters priors into one integral over the parameter domain.

Lemma 3.1. *For all Stan programs p with $stmt = model(p)$ and $\mathcal{P} = params(p)$, and environments γ:*

$$\{\!|C(p)|\!\}_\gamma \propto \lambda U. \int_U \{\!|C_{\texttt{return(())}}\,(stmt)|\!\}_{\gamma[\mathcal{P}\leftarrow\theta]}(\{()\})d\theta$$

where $U \in \Sigma_X$ is a measurable set of parameter values, with $X = Dom(\mathcal{P})$.

The proof relies on the fact that the parameters are sampled from the distributions `uniform` or `improper_uniform` which both have constant density w.r.t. the Lebesgue measure on their domain. These constants introduce a constant ratio ($\propto$) between the two measures. In addition, since parameters cannot appear on the left-hand side of an assignment we can simplify the `return` statement. The detailed proof is given in [1].

Score and `target`. We now show that the value of the Stan `target` variable corresponds to the score computed by the generated code.

Lemma 3.2. *For all Stan statements stmt compiled with a continuation k, if $\gamma(\texttt{target}) = 0$, and $[\![stmt]\!]_\gamma = \gamma'$,*

$$\{\!|C_k\,(stmt)|\!\}_\gamma = \lambda U.\, \exp(\gamma'(\texttt{target})) \times \{\!|k|\!\}_{\gamma'[\texttt{target}\leftarrow 0]}(U)$$

The proof is done by induction on the structure of *stmt* and the finite number of loops iterations (Assumption 1). The hypothesis $\gamma(\texttt{target}) = 0$ simplifies the induction by avoiding to keep an accumulator of the value of `target`. Resetting the value of target in the environment $\gamma'[\texttt{target} \leftarrow 0]$ for the evaluation of the continuation k is thus necessary for the inductive step. The proof is given in [1].

Correctness. We now have all the elements to prove that the comprehensive compilation is correct. That is, generated code yields the same un-normalized measure up to a constant factor that will be normalized away by the inference.

Theorem 3.3. *For all Stan programs p, the semantics of the source and compiled programs are equal up to a constant:*

$$\{\!|p|\!\}_D \propto \{\!|C(p)|\!\}_D$$

Proof. The proof is a direct consequence of Lemmas 3.1 and 3.2 and the definition of the two semantics. With $stmt = model(p)$ and $\mathcal{P} = params(p)$:

$$\begin{aligned}
&\{\!|C(p)|\!\}_D \propto \lambda U. \int_U \{\!|C_{\texttt{return(())}}\,(stmt)|\!\}_{D[\mathcal{P}\leftarrow\theta]}(\{()\})\, d\theta \\
&= \lambda U. \int_U \exp([\![stmt]\!]_{D[\mathcal{P}\leftarrow\theta]}\,(\texttt{target})) \times \{\!|\texttt{return(())}|\!\}(\{()\})\, d\theta \\
&= \lambda U. \int_U \exp([\![stmt]\!]_{D[\mathcal{P}\leftarrow\theta]}\,(\texttt{target}))\, d\theta \;=\; \{\!|p|\!\}_D
\end{aligned}$$

□

4 Implementation

We implemented two new backends for the Stan compiler targeting Pyro [3] and NumPyro [25]. NumPyro is a variant of Pyro built on top of JAX [5], a Python library that provides efficient automatic differentiation, vectorization, and just-in-time compilation on CPU, GPU, and TPU.

For both backends, we have implemented three compilation schemes: generative (Section 2.1), comprehensive (Section 2.3), and mixed.

Mixed compilation. The *mixed* compilation scheme is an optimization of the comprehensive translation where proper priors are used whenever possible. The mixed compilation can thus generate code that is similar to the generative translation whenever possible.

The mixed translation can be decomposed into three steps: first, compile the program with the comprehensive scheme; second, using the commutativity theorem of [30], reschedule `sample`(*uniform*) statements as late as possible and reschedule `observe`(D, x) statements as early as possible; and third, merge consecutive `sample` and `observe` statements using the following property:

$$\texttt{let}\ x = \texttt{sample}(\textit{uniform})\ \texttt{in}\ \texttt{let}\ () = \texttt{observe}(D, x)\ \texttt{in}\ e \equiv \texttt{let}\ x = \texttt{sample}(D)\ \texttt{in}\ e$$

In Stan, distribution are automatically truncated based on the parameter support as in the following model.

```
parameters { real<lower=0> sigma; }
model { sigma ~ normal(0, 1); }
```

The merge between the `sample` and `observe` statements is thus only correct if the two distributions have the same support. We extended the signature of Stan distributions to include the definition domain which can be used to check if the merge is possible.

Finally, the mixed compilation generates correct code even if the ~ statements do not respect the dependency order such as in the following example.

```
y ~ normal(x, 1); x ~ normal(0, 1); ...
```

The mixed compilation reschedules the statements to generate the following code which does not break any environment update:

```
let x = sample(normal(0, 1)) in
let y = sample(normal(x, 1)) in ...
```

Architecture. We implemented the compiler as a fork of the Stanc3 compiler[1], thus guarantying compatibility with the official Stan syntax and existing static analyses. The Stanc3 compiler is composed of multiple intermediate languages. We decided to implement the new backends for the first internal language which is the closest to the Stan source. The implementation is thus closer to the formalization, making it easier to keep track of the correspondence between the Stan source and the generated Pyro and NumPyro code. In particular, in Pyro and NumPyro all sampling sites (corresponding to `sample` and `observe`) are associated to a unique name which can be used for diagnostics and results analyses. We use Stan variable names with an optional postfix to preserve uniqueness when necessary. For example, in loops, the postfix tracks the current iteration.

Compiling to Pyro. The compiler addresses common challenges like name handling. Pyro and Stan naming conventions are different (e.g., `lambda` is a common parameter name in Stan) and Pyro has a shared namespace, while Stan distinguishes variables and functions. The compiler carefully avoids conflicts by renaming. Moreover, Stan supports function overloading. The compiler uses static type information to disambiguate function calls by renaming. Finally, there are semantics differences like one-based vs. zero-based arrays.

Stan has a large standard library that also has to be ported to Pyro. Our implementation currently supports a substantial portion of, but not the entire, standard library. Even though Pyro is built on top of Python and thus benefits from a large set of packages, it is not straightforward to implement all Stan functions. The Python counterpart sometimes also has typing or semantic differences. For example, the Stan Bernoulli distribution returns an integer and the Pyro one a float. The differences are handled either in the library or in the compiler. The categorical distribution, which is defined on $[1, N]$ in Stan and on $[0, N-1]$ in Pyro, illustrates both aspects. The translation of categories is done in the library for a call to the log probability mass function:

```
def categorical_lpmf(y, theta):
  return Categorical(theta).log_prob(y - 1)
```

and the compiler translates (`y ~ categorical(theta)`) into

```
observe(categorical(theta), y - 1)
```

Pyro does not require type declarations, but preserving the shape information of the indexes structures (arrays, vectors, row vectors, matrices) is important. Parameter shapes are passed as arguments to the priors' initialization.

Finally, compared to GProb, Pyro is a Python library. The compiler can thus use Python imperative features and side-effects for in-place mutation of arrays. However, inference cannot run on models with arbitrary side-effects. For instance, we need to introduce explicit copies when array cells are updated inside loops.

Compiling to NumPyro. The NumPyro backend shares the Pyro backend's challenges and has additional constraints coming from JAX. Dynamic features like dynamic array slices are not supported and will fail during inference. Control structures (conditional and loops) are library functions where the body must be passed in as a pure function. Our compiler accomplishes this by lambda-lifting the bodies of the control structures. This is similar to the compilation of

[1] https://github.com/stan-dev/stanc3

Stan loops to GProb described in Section 3.3 where the updated variables are given explicitly. Returning to the coin example (Figure 1), the compiled NumPyro code using the mixed compilation scheme is thus

```
def model(N, x):
  z = sample(beta(1, 1))
  def fori__2(i, acc):
    observe(bernoulli(z), x[i - 1])
  _ = fori_loop(1, N + 1, fori__2, None)
```

NumPyro loops are a recent feature which can have a noticeable performance impact. If the body of a loop does not contain probabilistic constructs, we thus generate a JAX loop instead of a NumPyro one.

5 Extending Stan: Explicit Variational Guides and Neural Networks

Probabilistic languages like Pyro offer new features to program and reason about complex models. This section shows that our compilation scheme can be used to lift these benefits for Stan users. Building on Pyro, we propose DeepStan, a conservative extension of Stan with: (1) variational inference with high-level but explicit guides, and (2) a clean interface to neural networks written in PyTorch.

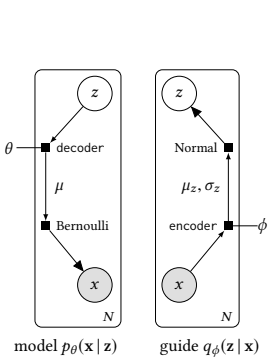
5.1 Explicit Variational Guides

Variational Inference (VI) tries to find the member $q_{\theta^*}(z)$ of a family $\mathcal{Q} = \{q_\theta(z)\}_{\theta\in\Theta}$ of simpler distributions that is the closest to the true posterior $p(z \mid \mathbf{x})$ [4]. Members of the family $\mathcal{Q}$ are characterized by the values of the *variational parameters* θ. The fitness of a candidate is measured using the Kullback-Leibler (KL) divergence from the true posterior, which VI aims to minimize:

$$q_{\theta^*}(z) = \operatorname*{argmin}_{\theta\in\Theta} \mathrm{KL}\Big(q_\theta(z) \,\|\, p(z \mid \mathbf{x})\Big).$$

Pyro natively supports variational inference and lets users define the family $\mathcal{Q}$ (the *variational guide*) alongside the model. To support this for Stan users, we extend Stan with two new optional blocks: `guide parameters` and `guide`. The `guide` block defines a distribution parameterized by the `guide parameters`. Variational inference optimizes the values of these parameters to approximate the true posterior.

DeepStan inherits restrictions for the definition of the guide from Pyro: the guide must be defined on the same parameter space as the model, i.e., it must sample all the parameters of the model; and the guide should also describe a distribution from which we can directly generate valid samples without running the inference first, which prevents the use of non-generative features and updates of `target`. The generative translation from Section 2.1 generates a Python function that can serve as a Pyro guide. The `guide parameters` block is used to generate Pyro `param` statements, which introduce learnable parameters. Unlike Stan parameters that define random variables for use in the model, guide parameters are learnable coefficients that will be optimized during inference.




```
networks {
  real[,] decoder(real[] x);
  real[,] encoder(int[,] x); }
data {
  int nz;
  int<lower=0, upper=1> x[28, 28]; }
parameters {
  real z[nz]; }
model {
  real mu[28, 28];
  z ~ normal(0, 1);
  mu = decoder(z);
  x ~ bernoulli(mu); }
guide {
  real encoded[2, nz] = encoder(x);
  real mu_z[nz] = encoded[1];
  real sigma_z[nz] = encoded[2];
  z ~ normal(mu_z, sigma_z); }
```

Figure 8. Graphical models and DeepStan code of the Variational Auto-Encoder model and guide.

These restrictions still allow sophisticated guides. The following section presents a guide defined by a neural network.

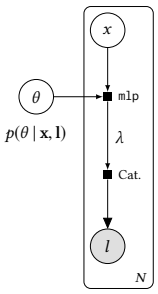
5.2 Adding Neural Networks

One important advantage of Pyro is its tight integration with PyTorch, enabling the authoring of *deep probabilistic models*: probabilistic models involving neural networks. It is impractical to define neural networks directly in Stan. To support deep probabilistic models, we extend Stan with an optional `networks` block to import neural network definitions.

Neural networks can be used to capture intricate dynamics between random variables. An example is the *Variational Auto-Encoder* (VAE) illustrated in Figure 8. A VAE learns a vector-space representation z for each observed data point x (e.g., the pixels of an image) [17, 27]. Each data point x depends on the latent representation z in a complex non-linear way, via a deep neural network: the *decoder*. The leftmost part of Figure 8 shows the corresponding graphical model. The output of the decoder is a vector μ that parameterizes a Bernoulli distribution over each dimension of x (e.g., each pixel associated to its probability of being in the image).

The key idea of the VAE is to use variational inference to learn the latent representation. The guide maps each x to a latent variable z via another neural network: the *encoder*. The middle part of Figure 8 shows the graphical model of the guide. The encoder returns, for each input x, the parameters μ_z and σ_z of a Gaussian distribution in the latent space. Inference tries to learn good values for the parameters θ and ϕ, simultaneously training the decoder and the encoder.

The right part of Figure 8 shows the corresponding code in DeepStan. A network is introduced similarly to an external function with its signature and must be implemented in PyTorch. The network can be used in subsequent blocks, in particular the `model` block and the `guide` block.




```
networks { vector mlp(real[,,] imgs); }
data {
 int batch_size; int nx; int nh; int ny;
 real <lower=0, upper=1> imgs[28,28,batch_size];
 int <lower=1, upper=10> labels[batch_size]; }
parameters {
  real mlp.l1.weight[nh, nx]; real mlp.l1.bias[nh];
  real mlp.l2.weight[ny, nh]; real mlp.l2.bias[ny]; }
model {
  vector[batch_size] lambda;
  mlp.l1.weight ~  normal(0, 1);
  mlp.l1.bias ~ normal(0, 1);
  mlp.l2.weight ~ normal(0, 1);
  mlp.l2.bias ~  normal(0, 1);
  lambda = mlp(imgs);
  labels ~ categorical_logit(lambda); }
guide parameters {
  real w1_mu[nh, nx]; real w1_sigma[nh, nx];
  real b1_mu[nh]; real b1_sigma[nh];
  real w2_mu[ny, nh]; real w2_sigma[ny, nh];
  real b2_mu[ny]; real b2_sigma[ny]; }
guide {
  mlp.l1.weight ~ normal(w1_mu, exp(w1_sigma));
  mlp.l1.bias   ~ normal(b1_mu, exp(b1_sigma));
  mlp.l2.weight ~ normal(w2_mu, exp(w2_sigma));
  mlp.l2.bias   ~ normal(b2_mu, exp(b2_sigma)); }
```

Figure 9. Graphical models and DeepStan code of the Bayesian MLP.

5.3 Bayesian Networks

Neural networks can also be treated as probabilistic models. A *Bayesian neural network* is a neural network whose learnable parameters (weights and biases) are random variables instead of concrete values [23]. Building on Pyro features, we make it easy for users to *lift* neural networks, i.e., replace concrete neural network parameters by random variables.

The left side of Figure 9 shows a simple classifier for handwritten digits based on a multi-layer perceptron (MLP) where all the parameters are lifted to random variables. Unlike the networks used in the VAE, the parameters (regrouped under the variable θ) are represented using a circle to indicate random variables. The inference starts from prior beliefs about the parameters and learns distributions that fit observed data. We then generate samples of concrete weights and biases to obtain an ensemble of as many MLPs as desired. The ensemble can vote for predictions and can quantify agreement.

The right of Figure 9 shows the corresponding code in DeepStan. We let users declare lifted neural network parameters in Stan's `parameters` block just like any other random variables. Network parameters are identified by the name of the network and a path, e.g., `mlp.l1.weight`, following PyTorch naming conventions. The `model` block defines `normal(0,1)` priors for the weights and biases of the two linear layers of the MLP. Then, for each image, the computed label follows a categorical distribution parameterized by the output of the network, which associates a probability to each of the ten possible values of the discrete random variable `label`. The `guide parameters` define μ and σ, and the `guide` block uses those parameters to propose normal distributions for the model parameters.

Compiling Bayesian neural networks. To lift neural networks, we use Pyro `random_module`, a primitive that takes a PyTorch network and a dictionary of distributions and turns the network into a distribution of networks where each parameter is sampled from the corresponding distribution. We treat network parameters as any other random variables and apply the comprehensive translation from Section 2.3. This translation initializes parameters with a uniform prior.

```
priors = {}
priors['l1.weight'] = improper_uniform(shape=[nh, nx])
... # priors of the other parameters
lifted_mlp = pyro.random_module('mlp', mlp, priors)()
```

Then, the Stan ~ statements in the `model` block are compiled into Pyro `observe` statements.

```
mlp_params = dict(lifted_mlp.named_parameters())
observe(normal(0, 1), mlp_params['l1.weight'])
```

It is also possible to mix probabilistic parameters and non-probabilistic parameters. Our translation only lifts the parameters that are declared in the `parameters` block by only adding those to the `priors` dictionary.

6 Evaluation

We presented our compilation scheme and described the implementation of two new backends for the Stanc3 compiler, targeting Pyro and NumPyro. This section evaluates our compilation scheme and the proposed extensions.

6.1 Compiling Stan to Pyro

First we focus on compiling classic Stan models to Pyro and NumPyro. We consider three questions:

RQ1: Can we compile and run all Stan models?
RQ2: What is the impact of the compilation on accuracy?
RQ3: What is the impact of the compilation on speed?

To answer these, we used two publicly available benchmark suites: the `example-models`[2] repository and PosteriorDB [35], a database of Stan models with corresponding data, reference posterior samples, and the configuration used to obtain these samples with Stan. The experiments were run on a Linux server with 64 cores (2.10GHz, 40GB RAM) with the latest version of Pyro (1.5.0), NumPyro (0.4.1), and cmdstanpy (0.9.67), without GPUs. The code of the experiments is available at https://github.com/deepppl/evaluation.

RQ1: Generality of the compilation. We run our compiler on the 541 models of the `example-models` repository. Stanc3 semantics checks reject 10 models. Out of the remaining 531, we were able to compile 522 models with the comprehensive and mixed compilation schemes for both the

[2] https://github.com/stan-dev/example-models

Table 2. Successful inference run for 98 PosteriorDB models.

	Compr.	Mixed	Gener.
Pyro	87	87	36
NumPyro	83	83	35

Pyro and NumPyro backends, but only 166 with the generative scheme. This further validates the need of our comprehensive translation. The 9 failures all involve truncations, a feature that is not natively supported in Pyro.

To test the inference, we run 1 iteration on the 98 pairs (models, data) of PosteriorDB that can be compiled with Stanc3. Table 2 presents results for the three compilation schemes: comprehensive, mixed, and generative.

The mixed optimization has no influence on the results. Failures with the Pyro backend are caused by missing standard library functions that are complicated to port to Pyro. As discussed in Section 4, the NumPyro backend relies on JAX, which limits what can be expressed in the model. The additional errors all involve dynamic features that are not supported in JAX.

As a baseline, we run the same experiment with the generative translation. As expected, compilation fails on 60 models. The additional runtime errors are the same as for the comprehensive and mixed translations.

RQ2: Accuracy. To evaluate inference accuracy we compare posterior distributions with the criteria used by regression tests for Stan:[3] For each parameter, we check if the error between the means is less than 30% of the standard deviation of the reference. For multidimensional parameters we check the same property for every component:

$$|\text{mean}(\theta_{\text{ref}}) - \text{mean}(\theta)| < 0.3\ \text{stddev}(\theta_{\text{ref}}).$$

PosteriorDB provides reference samples for 49 pairs (model, dataset). Using Stan with the same configuration (iterations, warmups, chains, thinning, seed), only 31 pairs pass the accuracy test and are thus valid baselines for our evaluation. We run using the Pyro and NumPyro implementation of NUTS with the same configuration[4], and compare the results with the reference posteriors (NUTS, the No U-Turn Sampler [16], is an optimized HMC and Stan's preferred inference method). Table 3 summarizes the results (additional results on the 49 examples are given in [1]).

Most of the models yield posterior distributions that match the reference. The four remaining errors are due to the following missing functions in our implementation of the standard library: `cov_exp_quad` (accel_gp and gp_regr) and ODE solvers (lotka_volterra and one_comp_mm_elim_abs). The mismatch (garch11) is due to a constraint that we do not know how to compile in Pyro/NumPyro (the domain of a parameter is constrained by the value of another one).

[3] https://github.com/stan-dev/performance-tests-cmdstan

[4] The configuration interface for NUTS in Pyro and NumPyro is similar to the CmdStanPy interface.

Table 3 shows that NUTS in Pyro is much slower than its NumPyro counterpart on our examples. Due to the high computational cost of running inference in Pyro (more than 100h for hmm_drive_0), we focus on the NumPyro backend to compare the different compilation schemes. Results show no difference between the comprehensive compilation scheme and the mixed version. When the generative translation is possible, the results also match except for one example (eight_schools_noncentered) due to a parameter constraint that is not propagated in the model (see Section 4).

As discussed in Section 4, the mixed compilation scheme can recover the code produced by the generative translation when possible. Table 3 shows that the extra priors introduced by the translation have no impact on the accuracy of the inference when using NUTS. However, these priors could play a critical role for other inference schemes, e.g., the importance sampling algorithm.

RQ3: Speed. To compare inference speed, Table 3 reports average runtime for Stan and NumPyro over five runs with varying seed values. For obvious computational cost reasons, we only report the duration of one Pyro run. As in RQ2, iterations, warmups, chains, and thinning configurations are given in PosteriorDB.

Table 3 shows that the runtime of DeepStan with the NumPyro backend is competitive with Stan under all three compilation schemes. In addition, runtime durations for the models compiled with the mixed, comprehensive, and generative scheme are almost identical when inference succeeds. These results indicate that the chosen compilation scheme has negligible influence on inference speed. Moreover, as shown in the last column, the NumPyro backend speeds up most benchmarks compared to the highly optimized Stan inference engine (geometric mean: 2.3x on 26 benchmarks).

Two examples (eight_schools-eight_schools_centered and arma11) are very sensitive to random seed variations in Stan and NumPyro (relative standard deviation $std/mean > 1$). For all other examples $std/mean \leq 0.1$.

Some of the benchmarks involve nested loops that are still experimental in NumPyro (arK, dogs, dogs_log, hmm_drive_0, hmm_example). For these example NumPyro is typically slower that Stan. If we exclude them we get an overall speedup of 3.8x on 21 benchmarks.

For Table 3 we pre-compiled the models to focus on inference time. The average compilation time for our compiler with both backends was 0.3s (std: 0.02) compared to 10.5s (std: 5.1) for Stan. The NumPyro backend thus outperforms Stan in both compilation and inference speed for these examples.

6.2 Stan Extensions

This section evaluates DeepStan, our extension with explicit variational guides and support for deep probabilistic programming with neural networks. We consider two questions:

Table 3. Comparing inference results with PosteriorDB references

			Pyro	NumPyro			
Model	Dataset	Stan	Compr.	Compr.	Mixed	Gener.	Speedup
accel_gp	mcycle_gp	✓ 00:18:22	✗	✗	✗	✗	
arK	arK	✓ 00:00:57	✓ 45:39:30	✓ 00:00:38	✓ 00:00:37	✓ 00:00:34	1.48
arma11	arma	✓ 00:01:36	✓ 02:19:07	✓ 00:00:42	✓ 00:21:46	✓ 00:18:53	2.26
dogs	dogs	✓ 00:01:06	✓ 29:12:04	✓ 00:06:22	✓ 00:06:12	✓ 00:06:09	0.17
dogs_log	dogs	✓ 00:00:32	✓ 23:56:20	✓ 00:03:43	✓ 00:03:34	✗	0.14
earn_height	earnings	✓ 00:01:18	✓ 01:07:03	✓ 00:00:15	✓ 00:00:15	✗	5.04
eight_schools_centered	eight_schools	✓ 00:00:05	✓ 00:27:09	✓ 00:00:07	✓ 00:00:06	✓ 00:00:06	0.69
eight_schools_noncentered	eight_schools	✓ 00:00:01	✓ 00:17:36	✓ 00:00:06	✓ 00:00:06	❍ 00:00:06	0.19
garch11	garch	✓ 00:00:17	❍ 20:20:05	❍ 00:02:07	❍ 00:02:04	✗	
gp_regr	gp_pois_regr	✓ 00:00:02	✗	✗	✗	✗	
hmm_drive_0	bball_drive_event_0	✓ 00:03:50	✓ 108:34:15	✓ 00:25:42	✓ 00:25:38	✗	0.15
hmm_example	hmm_example	✓ 00:00:28	✓ 08:57:44	✓ 00:01:02	✓ 00:01:02	✗	0.46
kidscore_interaction	kidiq	✓ 00:01:42	✓ 01:40:32	✓ 00:00:13	✓ 00:00:13	✗	7.80
kidscore_interaction_c2	kidiq_with_mom_work	✓ 00:00:10	✓ 00:08:58	✓ 00:00:06	✓ 00:00:06	✗	1.62
kidscore_mom_work	kidiq_with_mom_work	✓ 00:00:14	✓ 00:12:16	✓ 00:00:09	✓ 00:00:09	✗	1.66
kidscore_momhs	kidiq	✓ 00:00:05	✓ 00:12:05	✓ 00:00:06	✓ 00:00:06	✗	0.86
kidscore_momhsiq	kidiq	✓ 00:00:28	✓ 00:42:54	✓ 00:00:08	✓ 00:00:08	✗	3.32
kidscore_momiq	kidiq	✓ 00:00:13	✓ 00:30:28	✓ 00:00:07	✓ 00:00:07	✗	1.82
kilpisjarvi	kilpisjarvi_mod	✓ 00:00:59	✓ 12:12:26	✓ 00:00:21	✓ 00:00:21	✗	2.87
logearn_height	earnings	✓ 00:01:19	✓ 00:59:53	✓ 00:00:15	✓ 00:00:15	✗	5.29
logearn_height_male	earnings	✓ 00:03:45	✓ 01:36:57	✓ 00:00:23	✓ 00:00:23	✗	9.81
logearn_logheight_male	earnings	✓ 00:14:27	✓ 06:19:24	✓ 00:01:15	✓ 00:01:15	✗	11.59
logmesquite_logvas	mesquite	✓ 00:00:14	✓ 00:52:51	✓ 00:00:08	✓ 00:00:08	✗	1.84
lotka_volterra	hudson_lynx_hare	✓ 00:03:06	✗	✗	✗	✗	
mesquite	mesquite	✓ 00:00:15	✓ 00:59:55	✓ 00:00:08	✓ 00:00:08	✗	1.90
nes	nes1980	✓ 00:03:36	✓ 00:50:02	✓ 00:00:15	✓ 00:00:15	✗	14.02
nes	nes1976	✓ 00:06:59	✓ 00:54:46	✓ 00:00:20	✓ 00:00:21	✗	20.56
nes	nes1972	✓ 00:07:58	✓ 00:50:51	✓ 00:00:25	✓ 00:00:25	✗	19.07
nes	nes2000	✓ 00:02:41	✓ 00:56:39	✓ 00:00:13	✓ 00:00:13	✗	12.14
nes	nes1996	✓ 00:06:38	✓ 00:55:58	✓ 00:00:22	✓ 00:00:22	✗	18.35
one_comp_mm_elim_abs	one_comp_mm_elim_abs	✓ 00:16:10	✗	✗	✗	✗	

✓ match, ❍ mismatch, ✗ error. Durations are reported in HH:MM:SS format. Speedup = Stan / NumPyro Compr.

RQ4: Are explicit variational guides useful?
RQ5: For deep probabilistic models, how does DeepStan compare to hand-written Pyro code?

RQ4: Explicit guides. The *multimodal* example shown in Figure 10 is a mixture of two Gaussian distributions with different means but identical variances. The first two histograms of Figure 10 show that in both Stan and DeepStan, this example is particularly challenging for NUTS. Using multiple chains, NUTS finds the two modes, but the chains do not mix and the relative densities are incorrect. This is a known limitation of HMC.[5]

Stan also offers ADVI [19], an implementation of black-box VI where guides are automatically synthesized from the model using a full-rank (or mean-field) approximation. These choices imply that ADVI cannot approximate multimodal distribution as illustrated in the last histogram of Figure 10. On the other hand, using the custom variational guide presented in Figure 10, DeepStan with VI is able to find the two modes.

RQ5: Deep probabilistic models. As Stan lacks support for deep probabilistic models, it cannot be used as a baseline. Instead, we compare the performance of the compiled code with hand-written Pyro code on the VAE described in Section 5.2 and a simple Bayesian neural network.

Variational autoencoders were not designed as a predictive model but as a generative model to reconstruct images. Evaluating the performance of a VAE is thus non-obvious. We trained two VAEs on the MNIST dataset using VI: one hand-written in Pyro, the other written in DeepStan. For each image in the test set, the trained VAEs compute a latent

[5] https://mc-stan.org/users/documentation/case-studies/identifying_mixture_models.html

```
parameters {
  real cluster; real theta; }
model {
  real mu;
  cluster ~ normal(0, 1);
  if (cluster > 0) mu = 20;
  else mu = 0;
  theta ~ normal(mu, 1); }
guide parameters {
  real m1; real m2;
  real<lower=0> s1;
  real<lower=0> s2; }
guide {
  cluster ~ normal(0, 1);
  if (cluster > 0) theta ~ normal(m1, s1);
  else theta ~ normal(m2, s2); }
```

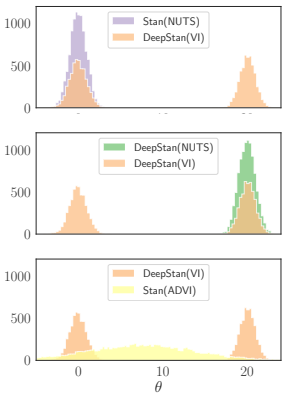


Figure 10. DeepStan code and histograms of the multimodal example using Stan, DeepStan with NUTS, DeepStan with VI, and Stan with ADVI.

representation of dimension 5. We cluster these representations using KMeans with 10 clusters. We measure VAE performance with the pairwise F1 metric: true positives are the number of images of the same digit that appear in the same cluster. For Pyro F1=0.41 (precision=0.43, recall=0.40), and for DeepStan F1=0.43 (precision=0.44, recall=0.42). These numbers shows that compiling DeepStan to Pyro does not impact the performance of such deep probabilistic models.

We trained two implementations of a 2-level Bayesian multi-layer perceptron (MLP) with the parameters all lifted to random variables (see section 5.3): one hand-written in Pyro, the other written in DeepStan. We trained both models for 20 epochs on the training set. For each model we generated 100 samples of concrete weights and biases to obtain an ensemble MLP that can be used to compute a distribution of predicted labels. The accuracy for both models is 92% on the test set and the agreement between the two models is above 95%. The execution time is comparable. These experiments show that compiling DeepStan models to Pyro has little impact on the model. Changing the priors on the network parameters from `normal(0,1)` to `normal(0,10)` (see Section 5.3) increases accuracy from 0.92 to 0.96. This further validates our compilation, compiling parameter priors to `observe` statements on deep probabilistic models.

7 Related Work

To the best of our knowledge, we propose the first comprehensive translation of Stan to a generative PPL. The closest related work was developed by the Pyro team [8]. Their work focuses on performance and ours on completeness. Their proposed compilation technique corresponds to the generative translation presented in Section 2.1 and thus only handles a subset of Stan. The code is not open-source, and we rely on our own implementation of the generative translation in Section 6. Compared to our approach, they are also looking into independence assumptions between loop iterations to generate parallel code. Combining these ideas with our approach is a promising future direction. They do not extend Stan with either VI or neural networks. Similarly, in Appendix B.2 of [15], Gorinova et al. outline the generative translation of Section 2.1, and also mention the issue with multiple updates but do not provide a solution. Lee et al. [20] introduce a density-based semantics for Pyro, but this semantics does not handle Stan's non-generative features.

The goal of compiling Stan to Pyro is to create a platform for experimenting with new ideas. For example, Section 5.1 extends Stan with explicit variational guides. Similarly, Pyro now offers inference on discrete parameters[6] that we could port to Stan using our backends.

In recent years, taking advantage of the maturity of DL frameworks, multiple deep probabilistic programming languages have been proposed: PyMC3 [28] built on top of Theano, Edward [33] and ZhuSuan [29] built on top of TensorFlow, and Pyro [3] and ProbTorch [22] built on top of PyTorch. All these languages are implemented as libraries. The users thus need to master the entire technology stack of the library, the underlying DL framework, and the host language. In comparison, DeepStan is a self-contained language and the compiler helps the programmer via dedicated static analyses.

8 Conclusion

This paper introduces a comprehensive compilation scheme from Stan to generative probabilistic programming languages. This shows that Stan is at most as expressive as this family of languages. We implemented a compiler from Stan to Pyro. Additionally, we designed and implemented extensions for Stan with explicit variational guides and neural networks.

Acknowledgements

The authors are greatful to the following people for their helpful feedback and encouragements: E. Bingham, K. Kate, Y. Mroueh, F. Obermeyer, A. Pauthier, and D. Phan.

References

[1] Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. 2021. Compiling Stan to Generative Probabilistic Languages and Extension to Deep Probabilistic Programming. *CoRR* abs/1810.00873 (2021).

[2] Guillaume Baudart, Martin Hirzel, and Louis Mandel. 2018. Deep Probabilistic Programming Languages: A Qualitative Study. *CoRR* abs/1804.06458 (2018).

[3] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6.

[6] https://pyro.ai/examples/enumeration.html

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

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[6] Bradley P Carlin and Thomas A Louis. 2008. *Bayesian methods for data analysis.* CRC Press.

[7] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–37. https://doi.org/10.18637/jss.v076.i01

[8] Jonathan P. Chen, Rohit Singh, Eli Bingham, and Noah Goodman. 2018. Transpiling Stan models to Pyro. In *ProbProg*.

[9] 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 *PLDI*. ACM, 221–236. https://doi.org/10.1145/3314221.3314642

[10] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models.* Cambridge university press. https://doi.org/10.1017/CBO9780511790942

[11] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis.* Chapman and Hall/CRC.

[12] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI*. AUAI Press, 220–229.

[13] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Accessed April 2021.

[14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *FOSE*. ACM, 167–181. https://doi.org/10.1145/2593882.2593900

[15] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2019. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *Proc. ACM Program. Lang.* 3, POPL (2019), 35:1–35:30. https://doi.org/10.1145/3290348

[16] Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

[17] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *ICLR*.

[18] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

[19] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2017. Automatic Differentiation Variational Inference. *J. Mach. Learn. Res.* 18 (2017), 14:1–14:45.

[20] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards verified stochastic variational inference for probabilistic programs. *PACMPL* 4, POPL (2020), 16:1–16:33. https://doi.org/10.1145/3371084

[21] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. 2009. The BUGS project: Evolution, critique and future directions. *Stat. in medicine* 28, 25 (2009), 3049–3067. https://doi.org/10.1002/sim.3680

[22] Siddharth Narayanaswamy, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank D. Wood, and Philip H. S. Torr. 2017. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. In *NIPS*. 5925–5935.

[23] Radford M. Neal. 1996. *Bayesian Learning for Neural Networks.* Vol. 118. Springer. https://doi.org/10.1007/978-1-4612-0745-0

[24] 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. In *AutoDiff Workshop*.

[25] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *CoRR* abs/1912.11554 (2019).

[26] Martyn Plummer et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Workshop on distr. stat. comp.*, Vol. 124. Vienna, Austria.

[27] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1278–1286.

[28] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Comput. Sci.* 2 (2016), e55. https://doi.org/10.7717/peerj-cs.55

[29] Jiaxin Shi, Jianfei Chen, Jun Zhu, Shengyang Sun, Yucen Luo, Yihong Gu, and Yuhao Zhou. 2017. ZhuSuan: A Library for Bayesian Deep Learning. *CoRR* abs/1709.05870 (2017).

[30] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 855–879. https://doi.org/10.1007/978-3-662-54434-1_32

[31] Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *LICS*. ACM, 525–534. https://doi.org/10.1145/2933575.2935313

[32] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *IFL*. ACM, 6:1–6:12. https://doi.org/10.1145/3064899.3064910

[33] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR (Poster)*.

[34] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018).

[35] Aki Vehtari and Måns Magnusson. 2020. PosteriorDB: a database with data, models and posteriors. In *Stan Conf.* https://github.com/stan-dev/posteriordb