Probabilistic Programming with Densities in SlicStan: Efficient, Flexible, and Deterministic

Probabilistic Programming with Densities in SlicStan: Efficient, Flexible, and Deterministic

MARIA I. GORINOVA, University of Edinburgh, UK
ANDREW D. GORDON, Microsoft Research Cambridge, UK and University of Edinburgh, UK
CHARLES SUTTON, Google Brain, US and University of Edinburgh, UK

Stan is a probabilistic programming language that has been increasingly used for real-world scalable projects. However, to make practical inference possible, the language sacrifices some of its usability by adopting a block syntax, which lacks compositionality and flexible user-defined functions. Moreover, the semantics of the language has been mainly given in terms of intuition about implementation, and has not been formalised.

This paper provides a formal treatment of the Stan language, and introduces the probabilistic programming language SlicStan — a compositional, self-optimising version of Stan. Our main contributions are (1) the formalisation of a core subset of Stan through an operational density-based semantics; (2) the design and semantics of the Stan-like language SlicStan, which facilities better code reuse and abstraction through its compositional syntax, more flexible functions, and information-flow type system; and (3) a formal, semantic-preserving procedure for translating SlicStan to Stan.

CCS Concepts: • **Theory of computation** → **Operational semantics**; **Program analysis**; *Probabilistic computation*; • **Mathematics of computing** → *Statistical software*; Markov-chain Monte Carlo methods;

Additional Key Words and Phrases: probabilistic programming, information flow analysis

ACM Reference Format:
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, Article 35 (January 2019), 30 pages. https://doi.org/10.1145/3290348

1 INTRODUCTION

1.1 Background: Probabilistic Programming Languages and Stan

Probabilistic programming languages [Gordon et al. 2014b] are a concise notation for specifying probabilistic models, while abstracting the underlying inference algorithm. There are many such languages, including BUGS [Gilks et al. 1994], JAGS [Plummer et al. 2003], Anglican [Wood et al. 2014], Church [Goodman et al. 2012], Infer.NET [Minka et al. 2014], Venture [Mansinghka et al. 2014], Edward [Tran et al. 2016] and many others.

Stan [Carpenter et al. 2017], with nearly 300,000 downloads of its R interface [Stan Development Team 2018a], is perhaps the most widely used probabilistic programming language. Stan's syntax is designed to enable automatic compilation to an efficient *Hamiltonian Monte Carlo* (HMC) inference algorithm [Neal et al. 2011], which allows programs to scale to real-word projects in statistics and data science. (For example, the forecasting tool Prophet [Taylor and Letham 2017] uses Stan.) This efficiency comes at a price: Stan's syntax lacks the compositionality of other similar systems, such as Edward [Tran et al. 2016] and PyMC3 [Salvatier et al. 2016]. The design of Stan assumes that the programmer needs to organise their model into separate blocks, which correspond to different stages of the inference algorithm (preprocessing, sampling, postprocessing). This compartmentalised

2475-1421/2019/1-ART35
https://doi.org/10.1145/3290348

syntax affects the usability of Stan: related statements may be separated in the source code, and functions are restricted to only acting within a single compartment. It is difficult to write complex Stan programs and encapsulate distributions and sub-model structures into re-usable libraries.

1.2 Goals and Key Insight

Our goals are (1) to examine the principles and assumptions behind the probabilistic programming language Stan that help it bridge the gap between probabilistic modelling and black-box inference; and (2) to design a suitable abstraction that captures the statistical meaning of Stan's compartments, but allows for compositional and more flexible probabilistic programming language syntax.

Our key insight is that *the essence of a probabilistic program in Stan is in fact a deterministic imperative program*, that can be automatically sliced into the different compartments used in the current syntax for Stan. It may come as a surprise that a probabilistic program is deterministic, but when performing Bayesian inference by sampling parameters, the probabilistic program serves to compute a deterministic score at a specific point of the parameter space. An implication of this insight is that standard forms of procedural abstraction are easily adapted to Stan.

1.3 The Insight by Example

As demonstration, and as a candidate for a future re-design of Stan, we present SlicStan[1] — a compositional, Stan-like language, which supports first-order functions.

Below, we show an example of a Stan program (right), and the same program written in SlicStan (left). In both cases, the goal is to obtain samples from the joint distribution of the variables $y \sim \mathcal{N}(0, 3)$ and $x \sim \mathcal{N}(0, \exp(y/2))$, using auxiliary standard normal variables for performance. Working with such auxiliary variables, instead of defining the model in terms of x and y directly, can facilitate inference and is a standard technique. We give more details in §§ 5.4 and Appendix D.

SlicStan

```
real my_normal(real m, real s) {
  real std ~ normal(0, 1);
  return s * std + m;
}
real y = my_normal(0, 3);
real x = my_normal(0, exp(y/2));
```

Stan

```
parameters {
  real y_std;
  real x_std;
}
transformed parameters {
  real y = 3 * y_std;
  real x = exp(y/2) * x_std;
}
model {
  y_std ~ normal(0, 1);
  x_std ~ normal(0, 1);
}
```

In both programs, the aim is to obtain samples for the random variables x and y, which are defined by scaling and shifting standard normal variables.

In SlicStan we do so by calling the function my_normal twice, which defines a local parameter std and encapsulates the transformation of each variable. Stan, on the other hand, does not support functions that declare new parameters, because all parameters must be declared inside the **parameters** block. We need to write out each transformation explicitly, also explicitly declaring each auxiliary parameter (x_std and y_std).

[1] SlicStan (pronounced *slick-Stan*) stands for "Slightly Less Intensely Constrained Stan".

The SlicStan code is a conventional deterministic imperative program, where the model statement `std ~ normal(0,1)` is a derived form of an assignment to a reserved variable that holds the score at a particular point of the parameter space. Due to the absence of blocks, SlicStan's syntax is compositional and more compact. Statements that would belong to different blocks of a Stan program can be interleaved, and no understanding about the performance implications of different compartments is required. Via an information-flow analysis, we automatically translate the program on the left to the one on the right.

1.4 Core Contributions and Outline

In short, this paper makes the following contributions:

- We formalise the syntax and semantics of a core subset of Stan (§ 2). To the best of our knowledge, this is the first formal treatment of Stan, despite the popularity of the language.
- We design SlicStan — a compositional Stan-like language with first-order functions. We formalise an information-flow type system that captures the essence of Stan's compartmentalised syntax, give the formal semantics of SlicStan, and prove standard results for the calculus, including noninterference and type-preservation properties (§ 3).
- We give a formal procedure for translating SlicStan to Stan, and prove that it is semantics preserving (§ 4).
- We examine the usability of SlicStan compared to that of Stan, using examples (§ 5).

The extended version of this paper [Gorinova et al. 2018a] includes an appendix (referred to as Appendix here) with additional details, discussion and examples.

2 CORE STAN

Stan [Carpenter et al. 2017] is a probabilistic programming language, whose syntax is similar to that of BUGS [Gilks et al. 1994; Lunn et al. 2013], and aims to be close to the model specification conventions used in the statistics community. This section gives the syntax (§§ 2.1; §§ 2.3) and semantics (§§ 2.2; §§ 2.4) of Core Stan, a core subset of Stan. The subset omits, for example, constraint data types, **while** loops, random number generators, recursive functions, and local variables.

To the best of our knowledge, our work is the first to give a formal semantics to the core of Stan. A full descriptive language specification can be found in the official reference manual [Stan Development Team 2017].

2.1 Syntax of Core Stan Expressions and Statements

The building blocks of a Stan statement are expressions. In Core Stan, expressions cover most of what the Stan manual specifies, including variables, constants, arrays and array elements, and function calls (of builtin functions). We let x range over variables. L-values are expressions limited to array elements $x[E_1]\dots[E_n]$, where the case $n = 0$ corresponds to a variable x. Statements cover the core functionality of Stan, with the exception of **while** statements, which we omit to to make *shredding* of SlicStan possible (see §§ 3.1 and §§ 4.1).

Core Stan Syntax of Expressions:

$E ::=$		expression
	x	variable
	c	constant
	$[E_1, ..., E_n]$	array
	$E_1[E_2]$	array element
	$f(E_1, \dots, E_n)$	function call[2]
$L ::= x[E_1]\dots[E_n]$	$n \geq 0$	L-value

Core Stan Syntax of Statements:

$S ::=$		statement
	$L = E$	assignment
	$S_1; S_2$	sequence
	for$(x$ **in** $E_1 : E_2)$ S	for loop
	if(E) S_1 **else** S_2	if statement
	skip	skip

We assume a set of builtin functions, ranged over by f. We also assume a set of standard builtin continuous or discrete distributions, ranged over by d. Each continuous distribution d has a corresponding builtin function d_lpdf, which defines its log probability density function. In this paper, we omit discrete random variables for simplicity.

Defined like this, the syntax of Stan statements is one of a standard imperative language. What makes the language *probabilistic* is the reserved variable **target**, which holds the logarithm[3] of the probability density function defined by the program (up to an additive constant), evaluated at the point specified by the current values of the program variables.

For example, to define a Stan model with random variables, μ and x, where we assume the variables are normally distributed and $\mu \sim \mathcal{N}(0, 1)$ and $x \sim \mathcal{N}(\mu, 1)$, we write:

```
target = normal_lpdf(mu, 0, 1) + normal_lpdf(x, mu, 1);
```
[4]

Here, normal_lpdf is the log density of the normal distribution: $\log \mathcal{N}(x|\, \mu, \sigma) = -\frac{(x-\mu)^2}{2\sigma^2} - \frac{1}{2}\log 2\pi\sigma^2$. The value of **target** is equal to the logarithm of the joint density over μ and x, $\log p(\mu, x)$, evaluated at the current values of the program variables mu and x. Suppose x is some known *data*, and μ is an unknown *parameter* of the model. We are interested in computing the *posterior distribution* of μ given x, $p(\mu \mid x) \propto p(\mu, x) = \mathcal{N}(x \mid \mu, 1)\mathcal{N}(\mu \mid 0, 1)$. Stan directly encodes a function that calculates the value of the log posterior density (up to an additive constant), and stores it in **target**. Thus, in addition to Stan's core statement syntax, we have a derived form for modelling statements:

Derived Form for Model Statements:

$E \sim \mathrm{d}(E_1, \ldots E_n) \triangleq$ **target** = **target** + d_lpdf$(E, E_1, \ldots E_n)$	model statement

In Stan, "~" is *not* considered to mean "draw a sample from", but rather "modify the joint distribution over parameters and data." This is also reflected by the semantics given in §§ 2.4.

2.2 Operational Semantics of Stan Statements

Next, we define a standard big-step operational semantics for Stan expressions and statements:

Big-step Relation

$(s, E) \Downarrow V$	expression evaluation
$(s, S) \Downarrow s'$	statement evaluation

Here, s and s' are states, and values V are the expressions conforming to the following grammar:

Values and States:

$V ::=$		value
$\quad c$		$\quad$ constant
$\quad [V_1, \ldots, V_n]$		$\quad$ array
$s ::= x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$	x_i distinct	state (finite map from variables to values)

The relation $\Downarrow$ is deterministic but partial, as we do not explicitly handle error states. The purpose of the operational semantics is to define a density function in §§ 2.4, and any errors lead to the density being undefined.

In the rest of the paper, we use the notation for states $s = x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$:

[2] If f is a binary operator, e.g. "+", we write it in infix.

[3] Stan evaluates the unnormalised density in the log domain to ensure numerical stability and to simplify internal computations. We follow this style throughout the paper, and define the semantics in terms of $\log p^*$, instead of p^*.

[4] We treat **target** as a mutable program variable for simplicity. This slightly differs from the actual implementation of Stan, where **target** does not allow for general lookup and update, but it is a special bit of state that can only be incremented.

- $s[x \mapsto V]$ is the state s, but where the value of x is updated to V if $x \in \text{dom}(s)$, or the element $x \mapsto V$ is added to s if $x \notin \text{dom}(s)$.
- $s[-x]$ is the state s, but where x is removed from the domain of s (if it were present).

We also define lookup and update operations on values:

- If U is an n-dimensional array value for $n \geq 0$ and $c_1, \ldots, c_n$ are suitable indexes into U, then the *lookup* $U[c_1] \ldots [c_n]$ is the value in U indexed by $c_1, \ldots, c_n$.
- If U is an n-dimensional array value for $n \geq 0$ and $c_1, \ldots, c_n$ are suitable indexes into U, then the *update* $U[c_1] \ldots [c_n] := V$ is the array that is the same as U except that the value indexed by $c_1, \ldots, c_n$ is V.

Operational Semantics of Expressions:

$$\frac{}{(s, c) \Downarrow c} \text{(Eval Const)} \qquad \frac{V = s(x) \quad x \in \text{dom}(s)}{(s, x) \Downarrow V} \text{(Eval Var)} \qquad \frac{(s, E_i) \Downarrow V_i \quad \forall i \in 1..n}{(s, [E_1, \ldots, E_n]) \Downarrow [V_1, \ldots, V_n]} \text{(Eval Arr)}$$

$$\frac{(s, E_1 \Downarrow V) \quad (s, E_2 \Downarrow c)}{(s, E_1[E_2]) \Downarrow V[c]} \text{(Eval ArrEl)} \qquad \frac{(s, E_i) \Downarrow V_i \quad \forall i \in 1 \ldots n \quad V = f(V_1, \ldots, V_n)^5}{(s, f(E_1, \ldots, E_n)) \Downarrow V} \text{(Eval PrimCall)}$$

Operational Semantics of Statements:

$$\frac{(s, E_i) \Downarrow V_i \quad \forall i \in 1..n \quad (s, E) \Downarrow V \quad U = s(x) \quad U' = (U[V_1] \ldots [V_n] := V)}{(s, L = E) \Downarrow (s[x \mapsto U'])} \text{(Eval Assign) (where } L = x[E_1] \ldots [E_n])$$

$$\frac{(s, S_1) \Downarrow s' \quad (s', S_2) \Downarrow s''}{(s, S_1; S_2) \Downarrow s''} \text{(Eval Seq)} \qquad \frac{(s, E) \Downarrow \textbf{true} \quad (s, S_1) \Downarrow s'}{(s, \textbf{if}(E)\ S_1\ \textbf{else}\ S_2) \Downarrow s'} \text{(Eval IfTrue)} \qquad \frac{(s, E) \Downarrow \textbf{false} \quad (s, S_2) \Downarrow s'}{(s, \textbf{if}(E)\ S_1\ \textbf{else}\ S_2) \Downarrow s'} \text{(Eval IfFalse)}$$

$$\frac{(s, E_1) \Downarrow c_1 \quad (s, E_2) \Downarrow c_2 \quad c_1 \leq c_2 \quad (s[x \mapsto c_1], S) \Downarrow s' \quad (s'[-x], \textbf{for}(x\ \textbf{in}\ (c_1 + 1) : c_2)\ S) \Downarrow s''}{(s, \textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S) \Downarrow s''} \text{(Eval ForTrue)}^6$$

$$\frac{(s, E_1) \Downarrow c_1 \quad (s, E_2) \Downarrow c_2 \quad c_1 > c_2}{(s, \textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S) \Downarrow s} \text{(Eval ForFalse)} \qquad \frac{}{(s, \textbf{skip}) \Downarrow s} \text{(Eval Skip)}$$

2.3 Syntax of Stan

A full Stan program consists of six program blocks, each of which is optional. Blocks appear in order. Each block has a different purpose and can reference variables declared in itself or previous blocks. Formally, we define a Stan program as a sequence of six blocks, each containing variable declarations or Stan statements, as shown next. We also present an example Stan program that contains all six blocks in §§ 5.2.

[5] $f(V_1, \ldots, V_n)$ means applying the builtin function f on the values $V_1, \ldots, V_n$.

[6] To make shredding to Stan possible, Core Stan only supports **for**-loops where the loop bounds do not change during execution: E_2 does not contain any variables that S writes to. This differs from the more flexible loops implemented in Stan.

Stan program:

$P ::=$	Stan Program
data { Γ_d }	
transformed data { Γ_{td}, S_{td} }	
parameters { Γ_p }	
transformed parameters { Γ_{tp}, S_{tp} }	
model { S_m }	
generated quantities { Γ_{gq}, S_{gq} }	

Arrays in Stan are sized, but we do not include any static checks on array sizes in this paper.

Stan Types and Type Environment:

$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n \quad \forall i \in 1 \ldots n \;\; x_i$ distinct	declarations
$\tau ::=$ **real** | **int** | **bool** | $\tau[n]$	type
n	size

The size of an array, n, can be a number or a variable. For simplicity, we treat n as decorative and do not include checks on the sizes in the type system of Stan. However, the system can be extended to a lightweight dependent type system, similarly to Tabular as extended by Szymczak [2018].

Each program block in Stan has a different purpose as follows:

- **data**: declarations of the input data.
- **transformed data**: definition of known constants and *preprocessing* of the data.
- **parameters**: declarations of the parameters of the model.
- **transformed parameters**: declarations and statements defining transformations of the data and parameters.
- **model**: statements defining the distributions of random variables in the model.
- **generated quantities**: declarations and statements that do not affect inference, used for *postprocessing*, or predictions for unseen data.

We define a conformance relation on states s and typing environments Γ. A state s conforms to an environment Γ, whenever s provides values of the correct types for the variables given in Γ:

Conformance Relation:

$s \models \Gamma$	state s conforms to environment Γ

Rule for the Conformance Relation:

(Stan State)

$$\frac{V_i \models \tau_i \quad \forall i \in I}{(x_i \mapsto V_i)^{i \in I} \models (x_i : \tau_i)^{i \in I}}$$

Here, $V \models \tau$ denotes that the value V is of type τ, and has the following definition:

- $c \models$ **int**, if $c \in \mathbb{Z}$, $c \models$ **real**, if $c \in \mathbb{R}$, and $c \models$ **bool** if $c \in \{$**true**, **false**$\}$.
- $[V_1, \ldots, V_n] \models \tau[m]$, if $\forall i \in 1 \ldots n. V_i \models \tau$.

We do not include any checks on array sizes in this paper, thus we do not assume n and m are the same in this definition. The evaluation relation is not defined on initial states that lead to array out-of-bounds errors.

2.4 Density-Based Semantics of Stan

Finally, we give the semantics of Stan in terms of the big-step relation from §§ 2.2. As the Stan Development Team [2017] explain:

> A Stan program defines a statistical model through a conditional probability function $p(\theta \mid y, x)$, where θ is a sequence of modeled unknown values (e.g., model parameters, latent variables, ...), y is a sequence of modeled known values, and x is a sequence of unmodeled predictors and constants (e.g., sizes, hyperparameters). (p. 22)

More specifically, a Stan program is executed to evaluate a function on the data and parameters $\log p^*(\boldsymbol{\theta} \mid \mathcal{D})$, for some given (and fixed) values of $\mathcal{D}$ and $\boldsymbol{\theta}$. This function encodes the log joint density of the data and parameters $\log p(\boldsymbol{\theta}, \mathcal{D})$ up to an additive constant, and also equals the log density of the posterior $\log p(\boldsymbol{\theta} \mid \mathcal{D})$ up to an additive constant:

$$\log p(\boldsymbol{\theta} \mid \mathcal{D}) = \log p(\boldsymbol{\theta}, \mathcal{D}) - \log p(\mathcal{D}) \propto \log p(\boldsymbol{\theta}, \mathcal{D}) \propto \log p^*(\boldsymbol{\theta} \mid \mathcal{D})$$

The return value of $\log p^*(\boldsymbol{\theta} \mid \mathcal{D})$ is stored in the reserved variable **target**. We give the semantics of Core Stan through this *unnormalised log posterior density function*.

Consider a Core Stan program P defined as previously, and the statement $S = S_{td}; S_{tp}; S_m; S_{gq}$. The semantics of P is the unnormalised log posterior density function $\log p^*$ on parameters $\boldsymbol{\theta}$ given data $\mathcal{D}$ (where $\boldsymbol{\theta} \models \Gamma_p$ and $\mathcal{D} \models \Gamma_d$):

$$\log p^*(\boldsymbol{\theta} \mid \mathcal{D}) \triangleq s'[\textbf{target}] \text{ if there is } s' \text{ such that } ((\mathcal{D}, \boldsymbol{\theta}, \textbf{target} \mapsto 0), S) \Downarrow s'$$

If there is no such s' then the log density is undefined. Observe also that if such an s' exists, it is unique, because the operational semantics is deterministic.

For example, suppose that P specifies a simple model for the data array y:

```
data { int N; real[N] y; }
parameters { real mu; real sigma; }
model {
  mu ~ normal(0, 1);
  sigma ~ normal(0, 1);
  for(i in 1:N){ y[i] ~ normal(mu, sigma); }
}
```

Suppose also that $\boldsymbol{\theta} = (\text{mu} \mapsto \mu, \text{sigma} \mapsto \sigma)$ and $\mathcal{D} = (\text{N} \mapsto n, \text{y} \mapsto \mathbf{y})$, for some μ, σ, n, and a vector $\mathbf{y}$ of length n. The statement $S = S_{td}; S_{tp}; S_m; S_{gq}$ is then the body of the **model** block as specified above. Then $((\mathcal{D}, \boldsymbol{\theta}, \textbf{target} \mapsto 0), S) \Downarrow s'$, with $s'[\textbf{target}] = \log \mathcal{N}(\mu \mid 0, 1) + \log \mathcal{N}(\sigma \mid 0, 1) + \sum_{i=1}^{n} \log \mathcal{N}(y_i \mid \mu, \sigma)$. This is precisely the log joint density on mu, sigma and y, which is proportionate to the posterior of mu and sigma given y.

The function $\log p^*(\boldsymbol{\theta} \mid \mathcal{D})$ is *not* a (log) density, but rather it encodes the logarithm of the density of the posterior up to an additive constant. Such unnormalised log density uniquely defines the log density $\log p(\boldsymbol{\theta} \mid \mathcal{D})$:

$$\log p(\boldsymbol{\theta} \mid \mathcal{D}) = \log p^*(\boldsymbol{\theta} \mid \mathcal{D}) - \log Z(\mathcal{D}) \text{ where } Z(\mathcal{D}) = \int p^*(\boldsymbol{\theta} \mid \mathcal{D})\, d\boldsymbol{\theta}$$

The value $Z(\mathcal{D})$ is called the *normalising constant* (it is a constant with respect to the variables $\boldsymbol{\theta}$ that the density is defined on). Computing $Z(\mathcal{D})$ is in most cases intractable. Thus, many inference algorithms (including those of Stan) are designed to successfully approximate the posterior, relying only on being able to evaluate a function proportional to it: an *unnormalised density function*, such as $\log p^*(\boldsymbol{\theta} \mid \mathcal{D})$ above.

The goal of this paper is to formalise the statistical meaning of a Stan program, as given by the quotation from the reference manual above. This semantics concentrates on defining the unnormalised log posterior of parameters given data, but omits the fact that the values of transformed parameters and generated quantities blocks are also part of the observable state. Transformed

parameters and generated quantities can be seen as variables that are generated using the function $g(\boldsymbol{\theta}, \mathcal{D}) \triangleq s'[\mathrm{dom}(\Gamma_{tp} \cup \Gamma_{gq})]$ for s' defined as previously. Appendix B.1 discusses generated quantities in more detail, and we leave their full treatment for future work. Moreover, Appendix B.2 discusses how this density-based semantics relates to other imperative probabilistic languages semantics, such as the sampling-based semantics of Prob [Hur et al. 2015].

2.5 Inference

Executing a Stan program consists of generating samples from the *posterior distribution* $p(\boldsymbol{\theta} \mid \mathcal{D})$, as a way of performing *Bayesian inference*. The primary algorithm that Stan uses is the asymptotically exact algorithm Hamiltonian Monte Carlo (HMC) [Neal et al. 2011], and more specifically, an enhanced version of the No-U-Turn Sampler (NUTS) [Betancourt 2017; Hoffman and Gelman 2014], which is an adaptive path lengths extension to HMC.

HMC is a Markov Chain Monte Carlo (MCMC) method (see Murray [2007] for a review on MCMC). Similarly to other MCMC methods, it obtains samples $\{\boldsymbol{\theta}_i\}_{i=1}^{\infty}$ from the target distribution, by using the latest sample $\boldsymbol{\theta}_n$ and a carefully designed transition function δ to generate a new sample $\boldsymbol{\theta}_{n+1} = \delta(\boldsymbol{\theta}_n)$. When sampling from the posterior $p(\boldsymbol{\theta} \mid \mathcal{D})$, HMC evaluates the unnormalised density $p^*(\boldsymbol{\theta} \mid \mathcal{D})$ at several points in the parameter space at each step n. To improve performance, HMC also uses the gradient of $\log p^*(\boldsymbol{\theta} \mid \mathcal{D})$ with respect to $\boldsymbol{\theta}$.

3 SLICSTAN

This section outlines the second key contribution of this work — the design and semantics of SlicStan. SlicStan is a probabilistic programming language, which aims to provide a more compositional alternative to Stan, while retaining Stan's efficiency and statement syntax natural to the statistics community. Thus, we design the language so that:

(1) SlicStan statements are a superset of the Core Stan statements given in § 2,
(2) SlicStan programs contain no program blocks, and allow the interleaving of statements that would belong to different program blocks if the program was written in Stan, and
(3) SlicStan supports first-order non-recursive functions.

This results in a flexible syntax, that allows for better encapsulation and code reuse, similarly to Edward [Tran et al. 2016] and PyMC3 [Salvatier et al. 2016].

The key idea behind SlicStan is to use *information flow analysis* to optimise and transform the program to Stan code. Secure information flow analysis has a long history, summarised by Sabelfeld and Myers [2003], and Smith [2007]. It concerns systems where variables have one of several *security levels*, and the aim is to disallow the flow of high-level information to a low-level variable, but allow other flows of information. For example, consider two security levels, public and secret. We want to forbid public information to depend on secret information. Formally, the levels form a *lattice* $(\{\text{public}, \text{secret}\}, <)$ with public $<$ secret. Secure information flow analysis is used to ensure that information flows only upwards in the lattice. This is formalized as the *noninterference property* [Goguen and Meseguer 1982]: confidential data may not interfere with public data.

Looking back to the description of Stan's program blocks in §§ 2.3, as well as the Stan Manual, we identify three information levels in Stan: data, model, and genquant. We assign one of these levels to each program block, as summarised by Table 1. 'Chain', 'sample' and 'leapfrog' refer to stages of the Hamiltonian Monte Carlo sampling algorithm. Usually, Stan runs several chains to perform inference, where there are many samples per chain, and many leapfrogs per sample.

Even though our insight about the three information levels comes from Stan, they are not tied to Stan's peculiarities. Variables at level data are the known quantities in the statistical inference problem, that is, the data. Computations at this level can be seen as a form of *preprocessing*.

Table 1. Program blocks in Stan. Adapted from Betancourt [2014].

Block	Execution	Level
data	—	DATA
transformed data	per chain	DATA
parameters	—	MODEL
transformed parameters	per leapfrog	MODEL
model	per leapfrog	MODEL
generated quantities	per sample	GENQUANT

Variables at level MODEL are unknown — they are the quantities we wish to infer. Changing the MODEL variables or the dependencies between them changes the statistical model we are working with, which can have a huge effect on the quality of inference. Finally, generated quantities are variables that DATA and MODEL variables do not depend on, and computing them can be seen as a form of *postprocessing*. All three are fundamental concepts of statistical inference and are not specific to Stan.

The rest of this section defines the SlicStan language. The syntax of SlicStan statements (§§ 3.1) extends that of the Core Stan statements from § 2, and its type system (§§ 3.2) assumes level types DATA, MODEL and GENQUANT on variables. The typing rules are then implemented so that in well-typed SlicStan programs, information flows from level DATA, through MODEL to GENQUANT. Every Core Stan program can be turned into an equivalent SlicStan program by concatenating the statements and declarations in its compartments.

Next, we give the semantics of a SlicStan program, much as we did for Core Stan, as an unnormalised log density function on parameters and data (§§ 3.4), and show some examples (§§ 3.5). To do so, we *elaborate* SlicStan's statements to Core Stan statements by statically unrolling user-defined function calls and bringing all variable declarations to the top level (§§ 3.3). The main purpose of elaboration is to identify all parameters statically so as to give the semantics as a function on the parameters. Elaboration also serves as a first step in translating SlicStan to Stan (§ 4).

3.1 Syntax

A SlicStan program is a sequence of function definitions F_i, followed by top-level statement S.

Syntax of a SlicStan Program

$F_1, \ldots, F_n, S \quad n \geq 0$	SlicStan program

SlicStan's user-defined functions are not recursive (a call to F_i can only occur in the body of F_j if $i < j$). Functions are specified by a return type T, arguments with their types $a_i : T_i$, and a body S. There is a reserved variable ret_g associated with each function, to hold the return value.

Syntax of Function Definitions

$F ::= T\ g(T_1\ a_1, \ldots, T_n\ a_n)\ S$	function definition (signature $g : T_1, \ldots, T_n \to T$)

SlicStan's expressions and statements extend those of Stan, by user-defined function calls $g(E_1, \ldots, E_n)$ and variable declarations $T\ x;\ S$ (in italic).

In both declarations $T\ x;\ S$ and loops **for**(x **in** $E_1 : E_2$) S, the variable x is locally bound with scope S. We identify statements up to consistent renaming of bound variables. Note that occurrences of variables in L-values are free.

SlicStan Syntax of Expressions:

$E ::=$	expression
x	variable
c	constant
$[E_1, ..., E_n]$	array
$E_1[E_2]$	array element
$f(E_1, \ldots, E_n)$	builtin function call
$g(E_1, \ldots, E_n)$	*user-defined fun. call*
$L ::=$	L-value
x	variable
$x[E_1] \ldots [E_n]$	array element

SlicStan Syntax of Statements:

$S ::=$	statement
$L = E$	assignment
$S_1; S_2$	sequence
for$(x$ **in** $E_1 : E_2)$ S	for loop
if(E) S_1 **else** S_2	if statement
skip	skip
$T\ x;\ S$	*declaration*

We constrain the language to only support **for** loops, disallowing the value of the loop guard to depend on the body of the loop. As described in later subsections, in order to give the semantics of a SlicStan program, as well as to translate it to Stan, we need to *elaborate* the statements to Core Stan statements (§§ 3.3), statically unrolling user-defined functions and extracting variable declarations to the top-level. Extending the language to support **while** loops (or recursive functions) means risk of non-terminating elaboration step, and a potentially inefficient resulting Stan program. This design choice is a small restriction on the usability and range of expressible models compared to Stan: models in Stan can only have a fixed number of parameters. As a result, an overwhelming number of examples in the Stan official repository use **for** loops only.

We define derived forms for data declarations, modelling statements, and return statements. Any user-defined function D_lpdf can be used as the log density function of a user-defined distribution D on the first argument of D_lpdf. For the sake of simplicity, we assume the body of a user-defined function g contains *at most one* return statement, at the end, and we treat it as an assignment to the return variable ret_g.

Derived Forms

data $\tau\ x; S \triangleq (\tau, \text{DATA})\ x; S$	data declaration
$E \sim \mathrm{d}(E_1, \ldots E_n) \triangleq$ **target** = **target** + d_lpdf$(E, E_1, \ldots E_n)$	model, builtin distribution
$E \sim \mathrm{D}(E_1, \ldots E_n) \triangleq$ **target** = **target** + D_lpdf$(E, E_1, \ldots E_n)$	model, user-defined distribution
return $E \triangleq$ ret_g $= E$	return

3.2 Typing of SlicStan

Next, we present SlicStan's type system. We define a lattice $(\{\text{DATA}, \text{MODEL}, \text{GENQUANT}\}, <)$ of level types, where DATA < MODEL < GENQUANT. Types T in SlicStan range over pairs (τ, ℓ) of a base type τ, and a level type ℓ — one of DATA, MODEL, or GENQUANT. Arrays are sized, with $n \geq 0$. Each builtin function f has a family of signatures $f : (\tau_1, \ell), \ldots, (\tau_n, \ell) \to (\tau, \ell)$, one for each level ℓ.

Types, and Typing Environment:

$\ell ::=$	level type
DATA	data, transformed data
MODEL	parameters, transformed parameters
GENQUANT	generated quantities
n	size
$\tau ::=$ **real** | **int** | **bool** | $\tau[n]$	base type
$T ::= (\tau, \ell)$	type: base type and level
$\Gamma ::= x_1 : T_1, \ldots, x_n : T_n \quad x_i$ distinct	typing environment

(While builtin functions of our formal system are level polymorphic, user-defined functions are monomorphic. This design choice was made to keep the system simple, and we see no challenges to polymorphism that are unique to SlicStan.)

We assume the type of the reserved **target** variable to be (**real**, MODEL): this variable can only be accessed within the **model** block in Stan, thus its level is MODEL. Each function g is associated with a single return variable ret_g matching the return type of the function.

Reserved variables

target : (**real**, MODEL)	log joint probability density function
ret_g : T	return value of a function $T\ g(T_1\ a_1, \ldots, T_n\ a_n)\ S$

We present the full set of declarative typing rules, inspired by those of the secure information flow calculus defined by Volpano et al. [1996], and more precisely, its summary by Abadi et al. [1999]. The information flow constraints are enforced by the subsumption rules (ESUB) and (SSUB), which together allow information to only flow upwards the DATA < MODEL < GENQUANT lattice.

Intuitively, we need to associate each expression E with a level type to prevent lower-level variables to *directly* depend on higher-level variables, such as in the case `d = m + 1`, for `d` of level DATA and `m` of level MODEL. We also need to associate each statement S with a level type to prevent lower-level variables to *indirectly* depend on higher-level variables, such as in the case **if**`(m > 0) d = 2`.

Judgments of the Type System:

$\Gamma \vdash E : (\tau, \ell)$	expression E has type (τ, ℓ) and reads only level ℓ and below
$\Gamma \vdash S : \ell$	statement S assigns only to level ℓ and above
$\vdash F$	function definition F is well-typed

The function **ty**`(c)` maps constants to their types (for example **ty**`(5.5)`= **real**).

Typing Rules for Expressions:

(ESUB) $\dfrac{\Gamma \vdash E : (\tau, \ell) \quad \ell \leq \ell'}{\Gamma \vdash E : (\tau, \ell')}$ (VAR) $\dfrac{}{\Gamma, x : T \vdash x : T}$ (CONST) $\dfrac{\textbf{ty}(c) = \tau}{\Gamma \vdash c : (\tau, \text{DATA})}$ (ARR) $\dfrac{\Gamma \vdash E_i : (\tau, \ell) \quad \forall i \in 1..n}{\Gamma \vdash [E_1, ..., E_n] : (\tau[n], \ell)}$

(ARREL) $\dfrac{\Gamma \vdash E_1 : (\tau[n], \ell) \quad \Gamma \vdash E_2 : (\textbf{int}, \ell)}{\Gamma \vdash E_1[E_2] : (\tau, \ell)}$ (PRIMCALL)$(f : T_1, \ldots, T_n \to T)$ $\dfrac{\Gamma \vdash E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash f(E_1, \ldots, E_n) : T}$ (FCALL)$(g : T_1, \ldots, T_n \to T)$ $\dfrac{\Gamma \vdash E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash g(E_1, \ldots, E_n) : T}$

Here and throughout, we make use of several functions on the language building blocks:

- $W(S)$ (Definition A.1) is the set of variables that are assigned to in S: $W(x = 2 * y) = \{x\}$.
- $R(S)$ (Definition A.2) is the set of variables read by S: $R(x = 2 * y) = \{y\}$.
- $\Gamma(L)$ (Definition A.3) is the type of the L-value L in the context Γ: $\Gamma(x[0]) =$ (**real**, DATA) for x : (**real**[], DATA) $\in \Gamma$.

The rule (DECL) for a variable declaration $(\tau, \ell)\ x; S$ has a side-condition ($x \notin \text{dom}(\Gamma)$), where Γ is the local typing environment, that enforces that the variable x is globally unique, that is, there is no other declaration of x in the program. The condition $x \notin W(S)$ in (FOR) enforces that the loop index x is immutable inside the body of the loop. In (SEQ), we make sure that the sequence $S_1; S_2$ is *shreddable*, through the predicate $\mathcal{S}(S_1, S_2)$ (Definition 4.7). This imposes a restriction on the range

of well-typed programs, which is needed both to allow translation to Stan (see §§ 4.1), and to allow interpreting of the program in terms of preprocessing, inference and postprocessing.

Using the rules for expressions and statements, we can also obtain rules for the derived statements.

Typing Rules for Statements:

$$\text{(SSub)}\ \frac{\Gamma \vdash S : \ell' \quad \ell \leq \ell'}{\Gamma \vdash S : \ell} \qquad \text{(Assign)}\ \frac{\Gamma(L) = (\tau, \ell) \quad \Gamma \vdash E : (\tau, \ell)}{\Gamma \vdash (L = E) : \ell} \qquad \text{(Decl)}\ \frac{\Gamma, x : (\tau, \ell) \vdash S : \ell' \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\tau, \ell)\, x; S : \ell'}$$

$$\text{(If)}\ \frac{\Gamma \vdash E : (\textbf{bool}, \ell) \quad \Gamma \vdash S_1 : \ell \quad \Gamma \vdash S_2 : \ell}{\Gamma \vdash \textbf{if}(E)\ S_1\ \textbf{else}\ S_2 : \ell} \qquad \text{(Seq)}\ \frac{\Gamma \vdash S_1 : \ell \quad \Gamma \vdash S_2 : \ell \quad \mathcal{S}(S_1, S_2)}{\Gamma \vdash (S_1; S_2) : \ell} \qquad \text{(Skip)}\ \frac{}{\Gamma \vdash \textbf{skip} : \ell}$$

$$\text{(For)}\ \frac{\Gamma \vdash E_1 : (\textbf{int}, \ell) \quad \Gamma \vdash E_2 : (\textbf{int}, \ell) \quad \Gamma, x : (\textbf{int}, \ell) \vdash S : \ell \quad x \notin \mathrm{dom}(\Gamma) \quad x \notin W(S)}{\Gamma \vdash \textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S : \ell}$$

Derived Typing Rules

$$\text{(DataDecl)}\ \frac{\Gamma, x : (\tau, \textsc{data}) \vdash S : \ell \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \textbf{data}\ \tau\ x; S : \ell} \qquad \text{(PrimModel)(d\_lpdf} : T, T_1, \ldots, T_n \to (\textbf{real}, \textsc{model}))\ \frac{\Gamma \vdash E : T \quad \Gamma \vdash E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash E \sim \mathrm{d}(E_1, \ldots E_n) : \textsc{model}}$$

$$\text{(Return)}\ \frac{\Gamma \vdash \mathrm{ret\_g} : (\tau, \ell) \quad \Gamma \vdash E : (\tau, \ell)}{\Gamma \vdash \textbf{return}\ E : \ell} \qquad \text{(FModel)(D} : T, T_1, \ldots, T_n \to (\textbf{real}, \textsc{model}))\ \frac{\Gamma \vdash E : T \quad \Gamma \vdash E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash E \sim \mathrm{D\_dist}(E_1, \ldots E_n) : \textsc{model}}$$

Finally, we complete the three judgments of the type system with the rule (FDef) for checking the well-formedness of a function definition. The condition $\ell_i \leq \ell$ ensures that the level of the result of a function call is no smaller than the level of its arguments.

Typing Rule for Function Definitions:

$$\text{(FDef)}\ \frac{a_1 : T_1, \ldots, a_n : T_n, \mathrm{ret\_g} : (\tau, \ell) \vdash S : \ell \quad T_i = (\tau_i, \ell_i) \quad \ell_i \leq \ell \quad \forall i \in 1..n}{\vdash (\tau, \ell)\ g(T_1\ a_1, \ldots, T_n\ a_n)\ S}$$

In our formal development, we implicitly assume a fixed program with well-typed functions $\vdash F_1, \ldots, \vdash F_n$. More precisely, we assume a given well-formed program defined as follows.

Well-Formed SlicStan Program:

A program $F_1, \ldots F_N, S$ is *well-formed* iff $\vdash F_1, \ldots, \vdash F_n$, and $\varnothing \vdash S : \textsc{data}$.

SlicStan statements are, by design, a superset of Core Stan statements. Thus, we can treat any Core Stan statement as a SlicStan statement with big-step operational semantics defined as in § 2. By extending the conformance relation $s \models \Gamma$ to correspond to a SlicStan typing environment, we can prove type preservation of the operational semantics, with respect to SlicStan's type system.

Rule of the Conformance Relation:

$$\text{(State)}\ \frac{V_i \models \tau_i \quad \forall i \in I}{(x_i \mapsto V_i)^{i \in I} \models (x_i : (\tau_i, \ell_i))^{i \in I}}$$

Theorem 3.1 (Type Preservation for $\Downarrow$).
For a Core Stan statement S and a Core Stan expression E:

(1) If $s \models \Gamma$ and $\Gamma \vdash E : (\tau, \ell)$ and $(s, E) \Downarrow V$ then $V \models \tau$.
(2) If $s \models \Gamma$ and $\Gamma \vdash S : \ell$ and $(s, S) \Downarrow s'$ then $s' \models \Gamma$.

Proof. By inductions on the size of the derivations of the judgments $(s, E) \Downarrow V$ and $(s, S) \Downarrow s'$. □

Finally, we state a termination-insensitive noninterference result. Intuitively, the result means that (observed) data cannot depend on the model parameters, and that generated quantities do not affect the log density distribution defined by the model.

Definition 3.2 (ℓ-equal states). Given a typing environment Γ, states $s_1 \models \Gamma$ and $s_2 \models \Gamma$ are ℓ-equal for some level ℓ (written $s_1 \approx_\ell s_2$), if they differ only for variables of a level strictly higher than ℓ:

$$s_1 \approx_\ell s_2 \triangleq \forall x : (\tau, \ell') \in \Gamma.\, (\ell' \leq \ell \implies s_1(x) = s_2(x))$$

Theorem 3.3 (Noninterference). *Suppose $s_1 \models \Gamma$, $s_2 \models \Gamma$, and $s_1 \approx_\ell s_2$ for some ℓ. Then for Core Stan statements S and Core Stan expressions E:*

(1) If $\Gamma \vdash E : (\tau, \ell)$ and $(s_1, E) \Downarrow V_1$ and $(s_2, E) \Downarrow V_2$ then $V_1 = V_2$.
(2) If $\Gamma \vdash S : \ell$ and $(s_1, S) \Downarrow s_1'$ and $(s_2, S) \Downarrow s_2'$ then $s_1' \approx_\ell s_2'$.

Proof. (1) follows by rule induction on the derivation $\Gamma \vdash E : (\tau, \ell)$, and using that if $\Gamma \vdash E : (\tau, \ell)$, $x \in R(E)$ and $\Gamma(x) = (\tau', \ell')$, then $\ell' \leq \ell$. (2) follows by rule induction on the derivation $\Gamma \vdash S : \ell$ and using (1). □

3.3 Elaboration of SlicStan

Similarly to Stan, a SlicStan program defines a probabilistic model, through an unnormalised log density function on the model parameters and data. That is, the semantics of SlicStan is in terms of a fixed (or data-dependent) number of variables. Therefore, in order to be able to formally give the semantics, we need to statically unroll calls to user-defined functions, and pull all variable declarations to the top level. (We discuss the difficulties of directly specifying the semantics of SlicStan without elaboration in §§ 3.6.)

We call this static unrolling step *elaboration*, and we formalise it through the elaboration relation $\Downarrow_\Gamma$. Intuitively, to elaborate a program $F_1, \ldots F_N, S$, we elaborate its main body S by unrolling any calls to $F_1, \ldots, F_N$ (as specified by (Elab FCall)), and move all variable declarations to the top level (as specified by (Elab Decl)). The result is an elaborated SlicStan statement S' and a list of variable declarations Γ. As mentioned previously, to avoid notational clutter, we assume a top-level SlicStan program $F_1, \ldots, F_N, S$. Since the syntax of a SlicStan statement differs from that of a Core Stan statement only by the presence of user-defined function calls and variable declarations, an *elaborated SlicStan* statement is also a well-formed *Core Stan* statement.

Elaboration Relation

$P \Downarrow_\varnothing \langle \Gamma,\ S' \rangle$	program elaboration
$S \Downarrow_\Gamma \langle \Gamma,\ S' \rangle$	statement elaboration
$E \Downarrow_\Gamma \langle \Gamma,\ S.E' \rangle$	expression elaboration
$F \Downarrow_\Gamma \langle r{:}T, A, \Gamma,\ S \rangle$	fun. definition elaboration

Elaboration Rule for a SlicStan Program

(Elab SlicStan)

$$\frac{S \Downarrow_\varnothing \langle \Gamma,\ S' \rangle}{F_1, \ldots F_N, S \Downarrow_\varnothing \langle \Gamma,\ S' \rangle}$$

The *unrolling* rule (Elab FCall) assumes a call to a user-defined function g with definition $F = T\ g(T_1\ a_1, \ldots, T_n\ a_n)\ S$, which elaborates as described by (Elab FDef).

Elaboration Rules for Expressions:

(Elab Var)
$$\frac{}{x \Downarrow_\Gamma \langle \varnothing,\ \textbf{skip}.x \rangle}$$

(Elab Const)
$$\frac{}{c \Downarrow_\Gamma \langle \varnothing,\ \textbf{skip}.c \rangle}$$

(Elab ArrEl)
$$\frac{E_1 \Downarrow_\Gamma \langle \Gamma_1,\ S_1.E_1' \rangle \quad E_2 \Downarrow_\Gamma \langle \Gamma_2,\ S_2.E_2' \rangle \quad \Gamma_1 \cap \Gamma_2 = \varnothing}{E_1[E_2] \Downarrow_\Gamma \langle \Gamma_1 \cup \Gamma_2,\ S_1; S_2.(E_1'[E_2']) \rangle}$$

(Elab Arr)
$$\frac{E_i \Downarrow_\Gamma \langle \Gamma_i,\ S_i.E_i' \rangle \quad \forall i \in 1..n \quad \bigcap_{i=1}^n \Gamma_i = \varnothing}{[E_1, ..., E_n] \Downarrow_\Gamma \langle \bigcup_{i=1}^n \Gamma_i,\ S_1; ...; S_n.([E_1', ..., E_n']) \rangle}$$

(Elab PrimCall)
$$\frac{E_i \Downarrow_\Gamma \langle \Gamma_i,\ S_i.E_i' \rangle \quad \forall i \in 1..n \quad \bigcap_{i=1}^n \Gamma_i = \varnothing}{f(E_1, ..., E_n) \Downarrow_\Gamma \langle \bigcup_{i=1}^n \Gamma_i,\ S_1; ...; S_n.f(E_1', ..., E_n') \rangle}$$

(Elab FCall) (where F is the definition for function g)
$$\frac{\begin{array}{c} E_i \Downarrow_\Gamma \langle \Gamma_{E_i},\ S_{E_i}.E_i' \rangle \quad \forall i \in 1..n \quad F \Downarrow_\Gamma \langle r_F : T_F, A_F, \Gamma_F,\ S_F \rangle \\ A_F = \{a_i : T_i \mid i \in 1..n\} \quad \{r_F : T_F\} \cap A_F \cap (\bigcap_{i=1}^n \Gamma_i) \cap \Gamma_F = \varnothing \end{array}}{g(E_1, ..., E_n) \Downarrow_\Gamma \langle \{r_F : T_F\} \cup A_F \cup (\bigcup_{i=1}^n \Gamma_i) \cup \Gamma_F,\ (S_{E_1}; a_1 = E_1'; ...; S_{E_n}; a_n = E_n'; S_F.r_F) \rangle}$$

Elaboration Rule for Function Definitions:

(Elab FDef)
$$\frac{S \Downarrow_{\{r:T\} \cup \Gamma_A \cup \Gamma} \langle \Gamma',\ S' \rangle \quad \Gamma_A = \{a_1 : T_1, ..., a_n : T_n\} \quad \{r\} \cap \mathrm{dom}(\Gamma_A) \cap \mathrm{dom}(\Gamma') = \varnothing}{T\ g(T_1\ a_1, ..., T_n\ a_n)\ S \Downarrow_\Gamma \langle r : T, \Gamma_A, \Gamma',\ S' \rangle}$$

As we identify statements up to α-conversion, $T\ x;\ x = 1$ elaborates to $\langle \{x_1 : T\},\ x_1 = 1 \rangle$, but also to $\langle \{x_2 : T\},\ x_2 = 1 \rangle$, and so on. The (Elab Decl) rule simply extracts a variable declaration to the top level. Other than recursively applying $\Downarrow_\Gamma$ to sub-parts of the statement, the (Elab If) and (Elab For) rules transform the guards of the respective compound statement to be the fresh variables g or g_1, g_2 respectively (as opposed to unrestricted expressions). This is a necessary preparation step needed for the program to be correctly translated to Stan later (see §§ 4.1 and Appendix C).

Elaboration Rules for Statements:

(Elab Decl)
$$\frac{S \Downarrow_{\{x:T\} \cup \Gamma} \langle \Gamma',\ S' \rangle \quad x \notin \mathrm{dom}(\Gamma')}{T\ x; S \Downarrow_\Gamma \langle \{x : T\} \cup \Gamma',\ S' \rangle}$$

(Elab Skip)
$$\frac{}{\textbf{skip} \Downarrow_\Gamma \langle \varnothing,\ \textbf{skip} \rangle}$$

(Elab Assign)
$$\frac{L \Downarrow_\Gamma \langle \Gamma_L,\ S_L.L' \rangle \quad E \Downarrow_\Gamma \langle \Gamma_E,\ S_E.E' \rangle \quad \Gamma_L \cap \Gamma_E = \varnothing}{L = E \Downarrow_\Gamma \langle \Gamma_L \cup \Gamma_E,\ S_L; S_E; L' = E' \rangle}$$

(Elab Seq)
$$\frac{S_1 \Downarrow_\Gamma \langle \Gamma_1,\ S_1' \rangle \quad S_2 \Downarrow_\Gamma \langle \Gamma_2,\ S_2' \rangle \quad \Gamma_1 \cap \Gamma_2 = \varnothing}{S_1; S_2 \Downarrow_\Gamma \langle \Gamma_1 \cup \Gamma_2,\ S_1'; S_2' \rangle}$$

(Elab If) (where $\Gamma \vdash E : T$)
$$\frac{E \Downarrow_\Gamma \langle \Gamma_E,\ S_E.E' \rangle \quad S_1 \Downarrow_\Gamma \langle \Gamma_1,\ S_1' \rangle \quad S_2 \Downarrow_\Gamma \langle \Gamma_2,\ S_2' \rangle \quad \{g : T\} \cap \Gamma_E \cap \Gamma_1 \cap \Gamma_2 = \varnothing}{\textbf{if}(E)\ S_1\ \textbf{else}\ S_2 \Downarrow_\Gamma \langle \{g : T\} \cup \Gamma_E \cup \Gamma_1 \cup \Gamma_2,\ (S_E; g = E'; \textbf{if}(g)\ S_1'\ \textbf{else}\ S_2') \rangle}$$

(Elab For) (where $\Gamma \vdash E_1 : T_1$ and $\Gamma \vdash E_2 : T_2$)
$$\frac{\begin{array}{c} E_1 \Downarrow_\Gamma \langle \Gamma_1,\ S_1.E_1' \rangle \quad E_2 \Downarrow_\Gamma \langle \Gamma_2,\ S_2.E_2' \rangle \quad S \Downarrow_{\Gamma \cup \{x:(\textbf{int}, \textsc{data})\}} \langle \Gamma_S,\ S' \rangle \\ \Gamma_V = v_\Gamma(\Gamma_S, n) \quad \{g_1 : T_1, g_2 : T_2, n : (\textbf{int}, \textsc{data})\} \cap \Gamma_1 \cap \Gamma_2 \cap \Gamma_V = \varnothing \end{array}}{\begin{array}{l} \textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S \Downarrow_\Gamma \langle \{g_1 : T_1, g_2 : T_2, n : (\textbf{int}, \textsc{data})\} \cup \Gamma_1 \cup \Gamma_2 \cup \Gamma_V, \\ S_1; S_2; g_1 = E_1'; g_2 = E_2'; n = g_2 - g_1 + 1; \textbf{for}(x\ \textbf{in}\ g_1 : g_2)\ v_S(x, \Gamma_V, S') \rangle \end{array}}$$

In some cases when elaborating a **for** loop, Γ_S will not be empty (in other words, the body of the loop will declare new variables). Thus, as (Elab For) shows, variables in Γ_S are upgraded to an

array, and then accessed by the index of the loop. We use the function v_S (Definition A.4) which takes a variable x, a typing environment Γ, and a statement S, and returns a statement S', where any mention of a variable $x' \in \mathrm{dom}(\Gamma)$ is substituted with $x'[x]$. For example, consider the statement `for(i in 1:N){real model x ~ normal(0,1); y[i] ~ normal(x,1);}` and an environment Γ, such that $\Gamma \vdash N : (\textbf{int}, \textsc{data})$. The body of the loop declares a new variable x, thus it elaborates to $\langle \Gamma_S,\ S' \rangle$, where $\Gamma_S = \{x : (\textbf{real}, \textsc{model})\}$, and $S' =$ `{x ~ normal(0,1); y[i] ~ normal(x,1);}`.

By (Elab For), $S \Downarrow_\Gamma \langle \{\mathtt{g1} : (\textbf{int}, \textsc{data}), \mathtt{g2} : (\textbf{int}, \textsc{data})\} \cup \Gamma_V,\ \textbf{for}(\mathtt{i}\ \textbf{in}\ \mathtt{g1{:}g2})\{S''\} \rangle$ where:

$$\Gamma_V = v_\Gamma(\Gamma_S, N) = \{x : (\textbf{real}[\mathtt{N}], \textsc{model})\}$$
$$S'' = v_S(i, \Gamma_V, S') = \mathtt{x[i]} \sim \mathtt{normal(0,1);}\ \ \mathtt{y[i]} \sim \mathtt{normal(x[i],1);}$$

Next, we state and prove type preservation of the elaboration relation.

Theorem 3.4 (Type preservation of $\Downarrow_\Gamma$). *For SlicStan statements S, SlicStan expressions E, and SlicStan function definitions F:*

(1) If $\Gamma \vdash E : (\tau, \ell)$ and $E \Downarrow_\Gamma \langle \Gamma',\ S'.E' \rangle$ then $\Gamma, \Gamma' \vdash S' : \textit{data}$ and $\Gamma, \Gamma' \vdash E' : (\tau, \ell)$.
(2) If $\Gamma \vdash S : \ell$ and $S \Downarrow_\Gamma \langle \Gamma',\ S' \rangle$ then $\Gamma, \Gamma' \vdash S' : \ell$
(3) If $F \Downarrow_\Gamma \langle \Gamma',\ S'.ret \rangle$ then $\Gamma, \Gamma' \vdash S' : \textit{data}$

Proof. By inductions on the size of the derivations of the judgments $E \Downarrow_\Gamma \langle \Gamma',\ S'.E' \rangle$, $S \Downarrow_\Gamma \langle \Gamma',\ S' \rangle$, and $F \Downarrow_\Gamma \langle \Gamma',\ S'.\mathrm{ret} \rangle$. □

3.4 Semantics of SlicStan

We now show how SlicStan's type system allows us to specify the semantics of the probabilistic program as an unnormalised posterior density function. This shows how the semantics of SlicStan connects to that of Stan, and demonstrates that explicitly encoding the roles of program variables into the block syntax of the language is not needed.

We specify the semantics — the unnormalised density $\log p^*_{F_1,\ldots,F_n,S}(\boldsymbol{\theta} \mid \mathcal{D})$ — in two steps.

3.4.1 Semantics of (Elaborated) SlicStan Statements. Consider an elaborated SlicStan statement S such that $\Gamma \vdash S : \textsc{data}$. The semantics of S is the function $\log p^*_{\Gamma \vdash S}$, such that for any state $s \models \Gamma$:

$$\log p^*_{\Gamma \vdash S}(s) \triangleq s'[\textbf{target}] \text{ if there is } s' \text{ such that } ((s, \textbf{target} \mapsto 0), S) \Downarrow s'$$

3.4.2 Semantics of SlicStan Programs. Consider a well-formed SlicStan program $F_1, \ldots, F_n, S$ and suppose that $S \Downarrow_\varnothing \langle \Gamma',\ S' \rangle$. (Observe that Γ' and S' are uniquely determined by $F_1, \ldots, F_n, S$.) Suppose also that:

- $\Gamma_\mathcal{D}$ corresponds to *data variables*, $\Gamma_\mathcal{D} = \{x : \ell \in \Gamma' \mid \ell = \textsc{data} \wedge x \notin W(S')\}$, and
- Γ_θ corresponds to *model parameters*, $\Gamma_\theta = \{x : \ell \in \Gamma' \mid \ell = \textsc{model} \wedge x \notin W(S')\}$.

Similarly to Stan (§§ 2.4), the semantics of a SlicStan program S is the unnormalised log posterior density function $\log p^*_{F_1,\ldots,F_n,S}$ on parameters $\boldsymbol{\theta}$ given data $\mathcal{D}$ (with $\boldsymbol{\theta} \models \Gamma_\theta$ and $\mathcal{D} \models \Gamma_\mathcal{D}$):

$$\log p^*_{F_1,\ldots,F_n,S}(\boldsymbol{\theta} \mid \mathcal{D}) \triangleq \log p^*_{\Gamma' \vdash S'}(\boldsymbol{\theta}, \mathcal{D}) \tag{1}$$

3.5 Examples

Next, we give two examples of SlicStan programs, their elaborated versions, and their semantics in the form of an unnormalised log density function. Here, we specify the levels of variables in SlicStan programs explicitly. In § 5 we describe how type inference can be implemented to infer optimal levels for program variables, thus making explicit declaration of levels unnecessary.

3.5.1 Simple Example. Consider a SlicStan program $\varnothing, S$ ($\varnothing$ denotes no function definitions), where we simply model the distribution of a data array y:

```
S = real MODEL mu ~ normal(0, 1);
    real MODEL sigma ~ normal(0, 1);
    int DATA N;
    real DATA y[N];
    for(i in 1:N){ y[i] ~ normal(mu, sigma); }
```

We define the semantics of S in three steps:

(1) Elaboration: $S \Downarrow_{\varnothing} \langle \Gamma', S' \rangle$, where:

$\Gamma' =$ mu : (**real**, MODEL), sigma : (**real**, MODEL), y : (**real**[N], MODEL), N : (**int**, DATA)

```
S' = mu ~ normal(0, 1);
     sigma ~ normal(0, 1);
     for(i in 1:N){
       y[i] ~ normal(mu, sigma); }
```

(2) Semantics of S': For any state $s \models \Gamma'$, $\log p^*_{\Gamma' \vdash S'}(s) = s'[\textbf{target}]$, where $(s, S') \Downarrow s'$. Thus:

$$\log p^*_{\Gamma' \vdash S'}(s) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + \textstyle\sum_{i=1}^{N} \log \mathcal{N}(y_i, \mu, \sigma)$$

(3) Semantics of S: We derive $\Gamma_{\mathcal{D}} = \{x : \ell \in \Gamma' \mid \ell = \text{DATA} \wedge x \notin W(S')\} = \{N, y\}$, and $\Gamma_{\theta} = \{x : \ell \in \Gamma' \mid \ell = \text{MODEL} \wedge x \notin W(S')\} = \{\mu, \sigma\}$. Therefore, the semantics of S is the unnormalised density on the parameters μ and σ, given data N and y:

$$\log p^*_S(\mu, \sigma \mid y, N) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + \textstyle\sum_{i=1}^{N} \log \mathcal{N}(y_i, \mu, \sigma)$$

3.5.2 User-defined Functions Example. Next, we look at an example that includes a user-defined function. Here, the function my_normal is a reparameterising function (§§ 5.4), that defines a Gaussian random variable, by scaling and shifting a standard Gaussian variable:

```
S = real MODEL my_normal(real MODEL m, real MODEL s){
      real MODEL x_std ~ normal(0, 1);
      return m + x_std * s;
    }
    real MODEL mu ~ normal(0, 1);
    real MODEL sigma ~ normal(0, 1);
    int DATA N;
    real GENQUANT x[N];
    for(i in 1:N) { x[i] = my_normal(mu, sigma); }
```

(1) Elaboration: $S \Downarrow_{\varnothing} \langle \Gamma', S' \rangle$, where:

$\Gamma' =$ mu : (**real**, MODEL), sigma : (**real**, MODEL), m : (**real**, MODEL), s : (**real**, MODEL), x_std : (**real**[N], MODEL), x : (**real**[N], GENQUANT), N : (**int**, DATA)

```
S' = mu ~ normal(0, 1);
     sigma ~ normal(0, 1);
     for(i in 1:N){
       m = mu; s = sigma;
       x_std[i] ~ normal(0, 1);
       x[i] = m + x_std[i] * s; }
```

(2) Semantics of S': Consider any $s \models \Gamma'$. Then:

$$\log p^*_{\Gamma' \vdash S'}(s) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + \textstyle\sum_{i=1}^{N} \log \mathcal{N}(x_i^{\text{std}}, 0, 1)$$

(3) Semantics of S: We derive $\Gamma_{\mathcal{D}} = \{N\}$, and $\Gamma_{\theta} = \{\mu, \sigma, \mathbf{x}^{\text{std}}\}$. The semantics of the program S is the unnormalised density on the parameters μ, σ, and $\mathbf{x}^{\text{std}}$, given data N:

$$\log p^*_S(\mu, \sigma, \mathbf{x}^{\text{std}} \mid N) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + \textstyle\sum_{i=1}^{N} \log \mathcal{N}(x_i^{\text{std}}, 0, 1)$$

3.6 Difficulty of Specifying Direct Semantics Without Elaboration

Specifying the direct semantics $\log p^*_{\varnothing \vdash S}(s)$, without an elaboration step, is not simple. SlicStan's user-defined functions are flexible enough to allow new model parameters to be declared inside of the body of a function. Having some of the parameters declared this way means that it is not obvious what the complete set of parameters is, unless we elaborate the program.

Consider the program from §§§ 3.5.2. Its semantics is $\log p^*_S(\mu, \sigma, \mathbf{x}^{\text{std}} \mid N) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + \sum_{i=1}^{N} \log \mathcal{N}(x_i^{\text{std}}, 0, 1)$. This differs from $\log p^*_S(\mu, \sigma, x^{\text{std}} \mid N)) = \log \mathcal{N}(\mu, 0, 1) + \log \mathcal{N}(\sigma, 0, 1) + N \times \log \mathcal{N}(x^{\text{std}}, 0, 1)$, which would be the accumulated log density in case we do not unroll the my_normal call, and instead implement direct semantics. In one case, the model has $N + 2$ parameters: $\mu, \sigma, x_1^{\text{std}}, \ldots, x_N^{\text{std}}$. In the other, the model has only 3 parameters: $\mu, \sigma, x^{\text{std}}$.

4 TRANSLATION OF SLICSTAN TO STAN

Translating SlicStan to Stan happens in two steps: *shredding* (§§ 4.1) and *transformation* (§§ 4.2). In this section, we formalise these steps and show that the semantics, seen as an unnormalised log posterior density function on parameters given data, is preserved in the translation.

4.1 Shredding

The first step in translating an elaborated SlicStan program to Stan is the idea of *shredding* (or *slicing*) by level. SlicStan allows statements that assign to variables of different levels to be interleaved. Stan, on the other hand, requires all DATA level statements to come first (in the **data** and **transformed data** blocks), then all MODEL level statements (in the **parameters**, **transformed parameters** and **model** blocks), and finally, the GENQUANT level statements (in the **generated quantities** block).

Therefore, we define the shredding relation $\Updownarrow_\Gamma$ on an elaborated SlicStan statement S and triples of *single-level statements* (S_D, S_M, S_Q) (Definition 4.1). That is, $\Updownarrow_\Gamma$ shreds a statement into three elaborated SlicStan statements S_D, S_M and S_Q, where S_D only assigns to variables of level DATA, S_M only assigns to variables of level MODEL, and S_Q only assigns to variables of level GENQUANT. We formally state and prove this result in Lemma 4.2.

Shredding Relation

$S \Updownarrow_\Gamma (S_D, S_M, S_Q)$	statement shredding

Currently, Stan can only assign to DATA variables inside the **transformed data** block, to MODEL variables inside the **transformed parameters** block, and to generated quantities inside the **generated quantities** block. Therefore, in Stan it is not possible to write an **if** statement or a **for** loop which assigns to variables of different levels inside its body. The (SHRED IF) and (SHRED FOR) rules resolve this by copying the entire body of the **if** statement or **for** loop on each of the three levels. Notice that we restrict the **if** and **for** guards to be variables (as opposed to any expression), which we have ensured is the case after the elaboration step ((ELAB IF) and (ELAB FOR)).

For example, consider the SlicStan program S, as defined below. It elaborates to S' and Γ', and it is then shredded to the single-level statements (S_D, S_M, S_Q):

```
S = real DATA d;
    real MODEL m;
    if(d > 0){
      d = 1;
      m = 2;
    }
```

```
Γ′ = {d : (real, DATA),
      m : (real, MODEL),
      g : (bool, DATA)}

S′ = g = (d > 0);
     if(g){d=1; m=2;}
```

```
S_D = g = (d > 0);
      if(g){d = 1;}
S_M = if(g){m = 2;}
S_Q = skip;
```

Shredding Rules for Statements:

$$\frac{\Gamma(L) = (\_, \textsc{data})}{L = E \Updownarrow_\Gamma (L = E, \textbf{skip}, \textbf{skip})}\ (\textsc{Shred DataAssign}) \qquad \frac{\Gamma(L) = (\_, \textsc{model})}{L = E \Updownarrow_\Gamma \textbf{skip}, L = E, \textbf{skip}}\ (\textsc{Shred ModelAssign}) \qquad \frac{\Gamma(L) = (\_, \textsc{genquant})}{L = E \Updownarrow_\Gamma \textbf{skip}, \textbf{skip}, L = E}\ (\textsc{Shred GenQuantAssign})$$

$$\frac{S_1 \Updownarrow_\Gamma S_{D_1}, S_{M_1}, S_{Q_1} \quad S_2 \Updownarrow_\Gamma S_{D_2}, S_{M_2}, S_{Q_2}}{S_1; S_2 \Updownarrow_\Gamma (S_{D_1}; S_{D_2}), (S_{M_1}; S_{M_2}), (S_{Q_1}; S_{Q_2})}\ (\textsc{Shred Seq}) \qquad \frac{}{\textbf{skip} \Updownarrow_\Gamma (\textbf{skip}, \textbf{skip}, \textbf{skip})}\ (\textsc{Shred Skip})$$

$$\frac{S_1 \Updownarrow_\Gamma (S_{D_1}, S_{M_1}, S_{Q_1}) \quad S_2 \Updownarrow_\Gamma (S_{D_2}, S_{M_2}, S_{Q_2})}{\textbf{if}(g)\ S_1\ \textbf{else}\ S_2 \Updownarrow_\Gamma (\textbf{if}(g)\ S_{D_1}\ \textbf{else}\ S_{D_2}), (\textbf{if}(g)\ S_{M_1}\ \textbf{else}\ S_{M_2}), (\textbf{if}(g)\ S_{Q_1}\ \textbf{else}\ S_{Q_2})}\ (\textsc{Shred If})$$

$$\frac{S \Updownarrow_\Gamma (S_D, S_M, S_Q)}{\textbf{for}(x\ \textbf{in}\ g_1 : g_2)\ S \Updownarrow_\Gamma (\textbf{for}(x\ \textbf{in}\ g_1 : g_2)\ S_D), (\textbf{for}(x\ \textbf{in}\ g_1 : g_2)\ S_M), (\textbf{for}(x\ \textbf{in}\ g_1 : g_2)\ S_Q)}\ (\textsc{Shred For})$$

In the rest of this section, we show that shredding a SlicStan program preserves its semantics (Theorem 4.9), in the sense that an elaborated program S has the same meaning as the sequence of its shredded parts $S_D; S_M; S_Q$. We do so by:

(1) Proving that shredding produces *single-level statements* (Definition 4.1 and Lemma 4.2).
(2) Defining a notion of *statement equivalence* (Definition 4.3) and specifying what conditions need to hold to change the order of two statements (Lemma 4.4).
(3) Showing how to extend the type system of SlicStan in order for the language to fulfil the criteria from (2) (Definition 4.7, Lemma 4.8).

Intuitively, a single-level statement of level ℓ is one that updates only variables of level ℓ.

Definition 4.1 (Single-level Statement $\Gamma \vdash \ell(S)$). S is a single-level statement of level ℓ with respect to Γ (written $\Gamma \vdash \ell(S)$) if and only if, $\Gamma \vdash S : \ell$ and $\forall x \in W(S)$ there is some τ, s.t. $x : (\tau, \ell) \in \Gamma$.

Lemma 4.2 (Shredding produces single-level statements).

$$S \Updownarrow_\Gamma (S_D, S_M, S_Q) \implies \Gamma \vdash \mathit{DATA}(S_D) \wedge \Gamma \vdash \mathit{MODEL}(S_M) \wedge \Gamma \vdash \mathit{GENQUANT}(S_Q)$$

The core of proving Theorem 4.9 is that if we take a statement S that is well-typed in Γ, and reorder its building blocks according to $\Updownarrow_\Gamma$, the resulting statement S' will be *equivalent* to S.

Definition 4.3 (Statement equivalence). $S \simeq S' \triangleq (\forall s, s'.(s, S) \Downarrow s' \iff (s, S') \Downarrow s')$

In the general case, to swap the order of executing S_1 and S_2, it is enough for each statement not to assign to a variable that the other statement reads or assigns to:

Lemma 4.4 (Statement Reordering). *For statements S_1 and S_2 that are well-typed in Γ, if $R(S_1) \cap W(S_2) = \varnothing$, $W(S_1) \cap R(S_2) = \varnothing$, and $W(S_1) \cap W(S_2) = \varnothing$ then $S_1; S_2 \simeq S_2; S_1$.*

Shredding produces single-level statements, therefore we only encounter reordering single-level statements of distinct levels. Thus, two of the conditions needed for reordering already hold.

Lemma 4.5. *If $\Gamma \vdash \ell_1(S_1), \Gamma \vdash \ell_2(S_2)$ and $\ell_1 < \ell_2$ then $R(S_1) \cap W(S_2) = \varnothing$ and $W(S_1) \cap W(S_2) = \varnothing$.*

To reorder the sequence $S_2; S_1$ according to Lemma 4.4, we need to satisfy one more condition, which is $R(S_2) \cap W(S_1) = \varnothing$. We achieve this through the predicate $\mathcal{S}$ in the (Seq) typing rule.

One way to define $\mathcal{S}(S_2, S_1)$ is so that it directly reflects this condition: $\mathcal{S}(S_2, S_1) = R(S_2) \cap W(S_1)$. This corresponds to a form of a single-assignment system, where variables become immutable once they are read.

We adopt a more flexible strategy, where we enforce variables of level ℓ to become immutable only once they have been *read at a level higher than* ℓ. We define:

- $R_{\Gamma \vdash \ell}(S)$: the set of variables x that are read at level ℓ in S. For example, if y is of level ℓ, then $x \in R_{\Gamma \vdash \ell}(y = x)$. (Definition A.5).
- $W_{\Gamma \vdash \ell}(S)$: the set of variables x of level ℓ that have been assigned to in S (Definition A.6).

Importantly, if $\Gamma \vdash \ell(S)$, then the sets $R_{\Gamma \vdash \ell}(S)$ and $W_{\Gamma \vdash \ell}(S)$ are the same as $R(S)$ and $W(S)$:

Lemma 4.6. *If* $\Gamma \vdash \ell(S)$, *then* $R_{\Gamma \vdash \ell}(S) = R(S)$ *and* $W_{\Gamma \vdash \ell}(S) = W(S)$.

Finally, we give the formal definition of $\mathcal{S}$:

Definition 4.7 (Shreddable sequence). $\mathcal{S}(S_1, S_2) \triangleq \forall \ell_1, \ell_2 . (\ell_2 < \ell_1) \implies R_{\Gamma \vdash \ell_1}(S_1) \cap W_{\Gamma \vdash \ell_2}(S_2) = \varnothing$

Lemma 4.8 (Commutativity of sequencing single-level statements).
If $\Gamma \vdash \ell_1(S_1), \Gamma \vdash \ell_2(S_2), \Gamma \vdash S_2; S_1 : \textsc{data}$ *and* $\ell_1 < \ell_2$ *then* $S_2; S_1; \simeq S_1; S_2;$

Theorem 4.9 (Semantic Preservation of $\Updownarrow_\Gamma$).
If $\Gamma \vdash S : \textsc{data}$ *and* $S \Updownarrow_\Gamma (S_D, S_M, S_Q)$ *then* $\log p^*_{\Gamma \vdash S}(s) = \log p^*_{\Gamma \vdash (S_D; S_M; S_Q)}(s)$, *for all* $s \models \Gamma$.

Proof. Note that if $S \simeq S'$ then $\log p^*_{\Gamma \vdash S}(s) = \log p^*_{\Gamma \vdash S'}(s)$ for all states $s \models \Gamma$. Semantic preservation then follows from proving the stronger result $\Gamma \vdash S : \textsc{data} \wedge S \Updownarrow_\Gamma (S_D, S_M, S_Q) \implies S \simeq (S_D; S_M; S_Q)$ by structural induction on the structure of S.

We give the full proof, together with proofs for Lemma 4.2, 4.4, 4.5 and 4.6, in A.2. □

4.2 Transformation

The last step of translating SlicStan to Stan is *transformation*. We formalise how a shredded SlicStan program $\langle \Gamma,\ (S_D, S_M, S_Q) \rangle$ transforms to a Stan program P, through the transformation relations:

Transformation Relations

$\Gamma \Downarrow_S^{(t)} P$	variable declarations transformation
$S \Downarrow^{(d)} P$	DATA statement transformation
$S \Downarrow^{(m)} P$	MODEL statement transformation
$S \Downarrow^{(q)} P$	GENQUANT statement transformation
$\langle \Gamma,\ S \rangle \Downarrow^{(t)} P$	top-level transformation

Intuitively, a shredded program $\langle \Gamma,\ (S_D, S_M, S_Q) \rangle$ transforms to Stan in four steps:

(1) The declarations Γ are split into blocks, depending on the level of variables and whether or not they have been assigned to inside of S_D, S_M or S_Q.
(2) The DATA-levelled statement S_D becomes the body of the **transformed data** block.
(3) The MODEL-levelled statement S_M is split into the **transformed parameters** and **model** block, depending on whether or not substatements assign to the **target** variable or not.
(4) The GENQUANT-levelled statement S_Q becomes the body of the **generated quantities** block.

This is formalised by the (Trans Prog) rule below. The Stan program $P_1; P_2$ is the Stan programs P_1 and P_2 merged by composing together the statements in each program block (Definition A.7).

Top-level Transformation Rule

$$\textsc{(Trans Prog)}\quad \frac{S \Uparrow_\Gamma (S_D, S_M, S_Q) \quad \Gamma \Downarrow^{(t)}_{(S_D;S_M;S_Q)} P \quad S_D \Downarrow^{(d)} P_D \quad S_M \Downarrow^{(m)} P_M \quad S_Q \Downarrow^{(q)} P_Q}{\langle \Gamma,\ S\rangle \Downarrow^{(t)} P; P_D; P_M; P_Q}$$

Transformation Rules for Declarations:

$$\textsc{(Trans Data)}\quad \frac{\Gamma \Downarrow^{(t)}_S P \qquad x \notin W(S)}{\Gamma, x : (\tau, \textsc{data}) \Downarrow^{(t)}_S \textbf{data}\{x : \tau\}; P}$$

$$\textsc{(Trans TrData)}\quad \frac{\Gamma \Downarrow^{(t)}_S P \qquad x \in W(S)}{\Gamma, x : (\tau, \textsc{data}) \Downarrow^{(t)}_S \textbf{transformed data}\{x : \tau\}; P}$$

$$\textsc{(Trans Param)}\quad \frac{\Gamma \Downarrow^{(t)}_S P \qquad x \notin W(S)}{\Gamma, x : (\tau, \textsc{model}) \Downarrow^{(t)}_S \textbf{parameters}\{x : \tau\}; P}$$

$$\textsc{(Trans TrParam)}\quad \frac{\Gamma \Downarrow^{(t)}_S P \qquad x \in W(S)}{\Gamma, x : (\tau, \textsc{model}) \Downarrow^{(t)}_S \textbf{transformed parameters}\{x : \tau\}; P}$$

$$\textsc{(Trans GenQuant)}\quad \frac{\Gamma \Downarrow^{(t)}_S P}{\Gamma, x : (\tau, \textsc{genquant}) \Downarrow^{(t)}_S \textbf{generated quantities}\{x : \tau\}; P}$$

$$\textsc{(Trans Empty)}\quad \frac{}{\varnothing \Downarrow^{(t)} \varnothing}$$

Transf. Rule for Data Statements:

$$\textsc{(Trans Data)}\quad \frac{}{S_D \Downarrow^{(d)} \textbf{transformed data}\{S_D\}}$$

Transf. Rule for GenQuant Statements:

$$\textsc{(Trans GenQuant)}\quad \frac{}{S_Q \Downarrow^{(d)} \textbf{generated quantities}\{S_Q\}}$$

The rules (Trans ParamIf), (Trans ModelIf), (Trans ParamFor), and (Trans ModelFor) might produce a Stan program that does not compile in the current version of Stan. This is because Stan restricts the **transformed parameters** block to only assign to transformed parameters, and the **model** block to only assign to the **target** variable. However, a **for** loop, for example, can assign to both kinds of variables in its body:

```
for(i in 1:N){
  sigma[i] = pow(tau[i], -0.5);
  y[i] ~ normal(0, sigma[i]); }
```

To the best of our knowledge, this limitation is an implementational particularity of the current version of the Stan compiler, and does not have an effect on the semantics of the language.[7] Therefore, we assume Core Stan to be a slightly more expressive version of Stan, that allows transformed parameters to be assigned in the **model** block.

Transformation Rules for Model Statements:

$$\textsc{(Trans ParamAssign)}\quad \frac{L \neq \textbf{target}}{L = E \Downarrow^{(m)} \textbf{transformed parameters}\{L = E\}}$$

$$\textsc{(Trans Model)}\quad \frac{}{\textbf{target} = E \Downarrow^{(m)} \textbf{model}\{\textbf{target} = E\}}$$

$$\textsc{(Trans ParamSeq)}\quad \frac{S_1 \Downarrow^{(m)} P_1 \quad S_2 \Downarrow^{(t)} P_2}{S_1; S_2 \Downarrow^{(m)} P_1; P_2}$$

[7] Moreover, there is an ongoing discussion amongst Stan developers to merge the parameters, transformed parameters and model blocks in future versions of Stan http://andrewgelman.com/2018/02/01/stan-feature-declare-distribute/.

$$\frac{\textbf{target} \notin W(S_1) \cup W(S_2)}{\textbf{if}(E)\, S_1\, \textbf{else}\, S_2 \Downarrow^{(m)} \textbf{transformed parameters}\{\textbf{if}(E)\, S_1\, \textbf{else}\, S_2\}} \quad \text{(Trans ParamIf)}$$

$$\frac{\textbf{target} \in W(S_1) \cup W(S_2)}{\textbf{if}(E)\, S_1\, \textbf{else}\, S_2 \Downarrow^{(m)} \textbf{model}\{\textbf{if}(E)\, S_1\, \textbf{else}\, S_2\}} \quad \text{(Trans ModelIf)}$$

$$\frac{\textbf{target} \notin W(S)}{\textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S \Downarrow^{(m)} \textbf{transformed parameters}\{\textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S\}} \quad \text{(Trans ParamFor)}$$

$$\frac{}{\textbf{skip} \Downarrow^{(m)} \varnothing} \quad \text{(Trans ParamSkip)}$$

$$\frac{\textbf{target} \in W(S)}{\textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S \Downarrow^{(m)} \textbf{model}\{\textbf{for}(x\ \textbf{in}\ E_1 : E_2)\ S\}} \quad \text{(Trans ModelFor)}$$

Theorem 4.10 (Semantic Preservation of $\Downarrow^{(t)}$). *Consider a well-formed SlicStan program $F_1, \ldots, F_n, S$, such that $S \Downarrow_{\varnothing} \langle \Gamma',\ S' \rangle$. Consider also a Core Stan program P, such that $\langle \Gamma',\ S' \rangle \Downarrow^{(t)} P$. Then for any $\boldsymbol{\theta} \models \{(x : (\tau, \textsc{data})) \in \Gamma' \mid x \notin W(S')\}$ and $\mathcal{D} \models \{(x : (\tau, \textsc{model})) \in \Gamma' \mid x \notin W(S')\}$:*

$$\log p^*_{F_1,\ldots,F_n,S}(\boldsymbol{\theta} \mid \mathcal{D}) = \log p^*_P(\boldsymbol{\theta} \mid \mathcal{D})$$

Proof. By rule induction on the derivation of $\langle \Gamma',\ S' \rangle \Downarrow^{(t)} P$, and equation 1 from §§§ 3.4.2. □

5 EXAMPLES AND DISCUSSION

In this section, we demonstrate and discuss the functionality of SlicStan. We compare several Stan code examples, from Stan's Reference Manual [Stan Development Team 2017] and Stan's GitHub repositories [Stan Development Team 2018b], with their equivalent written in SlicStan, and analyse the differences. All examples presented in this section have been tested using a preliminary implementation of SlicStan, developed by Gorinova et al. [2018b,c], although in this paper we use **for** loops where the work makes use of a vectorised notation.

Firstly, we assume a type inference strategy for level types, which allows us to remove the explicit specification of levels from the language (§§ 5.1). Next, we show that SlicStan allows the user to better follow the principle of locality — related concepts can be kept closer together (§§ 5.2). Secondly, we demonstrate the advantages of the more compositional syntax, when code refactoring is needed (§§ 5.3). The last comparison point shows the usage of more flexible user-defined functions, and points out a few limitations of SlicStan (§§ 5.4). More examples and a further discussion on the usability of the languages is presented in Appendix E.

5.1 Type Inference

Going back to §§ 2.3, and Table 1, we identify that different Stan blocks are executed a different number of times, which gives us another ordering on the level types: a performance ordering.

Code associated with variables of level DATA is executed only once, as a *preprocessing* step before inference. Code associated with variables of level GENQUANT is executed once per sample, right after inference has completed, as these quantities can be *generated* from the already obtained samples of the model parameters (in other words, this is a *postprocessing* step). Finally, code associated with MODEL variables is needed at each step of the inference algorithm itself. In the case of HMC, this means such code is executed once per leapfrog step (many times per sample).

Thus, there is a *performance ordering* of level types: DATA $\leq$ GENQUANT $\leq$ MODEL: it is cheaper for a variable to be DATA than to be GENQUANT, and is cheaper for it to be GENQUANT than to be MODEL. We can implement type inference following the rules from §§ 3.2, to infer the level type of each variable in a SlicStan program, so that:

- the hard constraint on the information flow direction DATA $<$ MODEL $<$ GENQUANT is enforced
- the choice of levels is optimised with respect to the ordering DATA $\leq$ GENQUANT $\leq$ MODEL.

We have implemented type inference for a preliminary version of SlicStan. In the rest of this section, we assume that no level type annotations are necessary in SlicStan, except for what the data of the probabilistic model is (specified using the derived form **data** $\tau\ x$; S), and that the optimal level type of each variable is inferred as part of the translation process.

5.2 Locality

With the first example, we demonstrate that SlicStan's blockless syntax makes it easier to follow good software development practices, such as declaring variables close to where they are used, and for writing out models that follow a *generative story*. It abstracts away some of the specifics of the underlying inference algorithm, and thus writing optimised programs requires less mental effort.

Consider an example adapted from [Stan Development Team 2017, p. 101]. We are interested in inferring the mean μ_y and variance σ_y^2 of the independent and identically distributed variables $\mathbf{y} \sim \mathcal{N}(\mu_y, \sigma_y)$. The model parameters are μ_y (the mean of $\mathbf{y}$), and $\tau_y = 1/\sigma_y^2$ (the precision of $\mathbf{y}$).

Below, we show this example written in SlicStan (left) and Stan (right).

SlicStan

```
real alpha = 0.1;
real beta = 0.1;
real tau_y ~ gamma(alpha, beta);

data real mu_mu;
data real sigma_mu;
real mu_y ~ normal(mu_mu, sigma_mu);

real sigma_y = pow(tau_y, −0.5);
real variance_y = pow(sigma_y, 2);

data int N;
data real[N] y;
for(i in 1:N){ y[i] ~ normal(mu_y, sigma_y);}
```

Stan

```
data {
  real mu_mu;
  real sigma_mu;
  int N;
  real y[N];
}
transformed data {
  real alpha = 0.1;
  real beta = 0.1;
}
parameters {
  real mu_y;
  real tau_y;
}
transformed parameters {
  real sigma_y = pow(tau_y,−0.5);
}
model {
  tau_y ~ gamma(alpha, beta);
  mu_y ~ normal(mu_mu, sigma_mu);
  for(i in 1:N){ y[i] ~ normal(mu_y, sigma_y);}
}
generated quantities {
  real variance_y = pow(sigma_y,2);
}
```

The lack of blocks in SlicStan makes it more flexible in terms of order of statements. The code here is written to follow more closely than Stan the *generative story*: we firstly define the prior distribution over parameters, and then specify how we believe data was generated from them.

We also keep declarations of variables close to where they have been used: for example, sigma_y is defined right before it is used in the definition of variance_y. This model can be expressed in SlicStan by using any order of the statements, provided that variables are not used before they are declared. In Stan this is not always possible and may result in closely related statements being located far away from each other.

With SlicStan there is no need to understand when different statements are executed in order to perform inference. The SlicStan code is translated to the hand-optimised Stan code, as specified by the manual, without any annotations from the user, apart from what the input data to the model is. In Stan, however, an inexperienced Stan programmer might have attempted to define the **transformed data** variables alpha and beta in the **data** block, which would result in a syntactic error. Even more subtly, they could have defined alpha, beta and variance_y all in the **transformed parameters** block, in which case the program will compile to a less efficient, semantically equivalent model.

5.3 Code Refactoring

The next example is adapted from [Stan Development Team 2017, p. 202], and shows how the absence of program blocks can lead to easier to refactor code. We start from a simple model, standard linear regression, and show what changes need to be made in both SlicStan and Stan, in order to change the model to account for measurement error. The initial model is a simple Bayesian linear regression with N predictor points $\mathbf{x}$, and N outcomes $\mathbf{y}$. It has 3 parameters — the intercept α, the slope β, and the amount of noise σ. In other words, $\mathbf{y} \sim \mathcal{N}(\alpha\mathbf{1} + \beta\mathbf{x}, \sigma I)$.

If we want to account for measurement noise, we need to introduce another vector of variables $\mathbf{x}_{meas}$, which represents the *measured* predictors (as opposed to the true predictors $\mathbf{x}$). We postulate that the values of $\mathbf{x}_{meas}$ are noisy (with standard deviation τ) versions of $\mathbf{x}$: $\mathbf{x}_{meas} \sim \mathcal{N}(\mathbf{x}, \tau I)$.

The next page shows these two models written in SlicStan (left) and Stan (right). Ignoring all the lines/corrections in red gives us the initial regression model, the one *not* accounting for measurement errors. The entire code, including the red corrections, gives us the second regression model, the one that *does* account for measurement errors. Transitioning from model one to model two requires the following corrections:

- **In SlicStan:**
  - Delete the **data** keyword for $\mathbf{x}$ (line 2).
  - Introduce *anywhere* in the program statements declaring the measurements $\mathbf{x}_{meas}$, their deviation τ, the now parameter $\mathbf{x}$, and its hyperparameters μ_x, σ_x (lines 11–17).
- **In Stan:**
  - Move $\mathbf{x}$'s declaration from **data** to **parameters** (line 5 and line 9).
  - Declare $\mathbf{x}_{meas}$ and τ in **data** (lines 3–4).
  - Declare $\mathbf{x}$'s hyperparameters μ_x and σ_x in **parameters** (lines 10–11).
  - Add statements modelling $\mathbf{x}$ and $\mathbf{x}_{meas}$ in **model** (lines 18–19).

Performing the code refactoring requires the same amount of code in SlicStan and Stan. However, in SlicStan the changes interfere much less with the code already written. We can add statements extending the model anywhere (as long variables are declared before they are used). In Stan, on the other hand, we need to modify each block separately. This example demonstrates a successful step towards our aim of making Stan more compositional — composing programs is easier in SlicStan.

Regression in SlicStan

```
data int N;
data real[N] x;
real mu_x;
real sigma_x;
data real[N] x_meas;
data real tau;

real alpha ~ normal(0, 10);
real beta ~ normal(0, 10);
real sigma ~ cauchy(0, 5);
data real[N] y;

for(i in 1:N){
  x[i] ~ normal(mu_x, sigma_x);
  x_mean[i] ~ normal(x[i], tau);
  y[i] ~ normal(alpha + beta*x[i], sigma);
}
```

Regression in Stan

```
data {
  int N;
  real[N] x_meas;
  real tau;
  real[N] x;
  real[N] y;
}
parameters {
  real[N] x;
  real mu_x;
  real sigma_x;
  real alpha;
  real beta;
  real sigma;
}
model {
  alpha ~ normal(0, 10);
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);
  for(i in 1:N){
    x[i] ~ normal(mu_x, sigma_x);
    x_mean[i] ~ normal(x[i], tau);
    y[i] ~ normal(alpha + beta*x[i], sigma); }
}
```

5.4 Code Reuse

Finally, we demonstrate the usage of more flexible functions in SlicStan, which allow for better code reuse, and therefore can lead to shorter, more readable code. In the introduction of this paper, we presented a transformation that is commonly used when specifying hierarchical model — the *non-centred parametrisation* of a normal variable. In brief, an MCMC sampler may have difficulties in exploring a posterior density well, if there exist strong non-linear dependencies between variables. In such cases, we can *reparameterise* the model: we can express it in terms of different parameters, so that the original parameters can be recovered. In the case of a normal variable $x \sim \mathcal{N}(\mu, \sigma)$, we define it as $x = \mu + \sigma x'$, where $x' \sim \mathcal{N}(0, 1)$. We explain in more detail the usage of the non-centered parametrisation in Appendix D.

In this section, we show the "Eight Schools" example [Gelman et al. 2013, p. 119], which also uses non-centred parametrisation in order to improve performance. Eight schools study the effects of their SAT-V coaching program. The input data is the estimated effects $\mathbf{y}$ of the program for each of the eight schools, and their shared standard deviation $\boldsymbol{\sigma}$. The task is to specify a model that accounts for errors, by considering the observed effects to be noisy estimates of the *true effects* $\boldsymbol{\theta}$. Assuming a Gaussian model for the effects and the noise, we have $\mathbf{y} \sim \mathcal{N}(\boldsymbol{\theta}, \boldsymbol{\sigma} I)$ and $\boldsymbol{\theta} \sim \mathcal{N}(\mu \mathbf{1}, \tau I)$.

Below is this model written in SlicStan (left) and Stan (right, adapted from Stan's GitHub repository [Stan Development Team 2018b]). In both cases, we use non-centred reparameterisation to improve performance: in Stan, the coaching effect for the i^{th} school, `theta[i]`, is declared as a transformed parameter obtained from the standard normal variable `eta[i]`; in SlicStan, we can once again make use of the non-centred reparameterisation function my_normal.

"Eight Schools" in SlicStan

```
real my_normal(real m, real v){
  real std ~ normal(0, 1);
  return v * std + m;
}

data real[8] y;
data real[8] sigma;
real[8] theta;

real mu;
real tau;

for (i in 1:8){
  theta[i] = my_normal(mu, tau);
  y[i] ~ normal(theta[i], sigma[i]);
}
```

"Eight Schools" in Stan

```
data {
  real y[8];
  real sigma[8];
}
parameters {
  real mu;
  real tau;
  real theta_std[8];
}
transformed parameters {
  real theta[8];
  for (j in 1:8){theta[j] = mu+tau*theta_std[i];}
}
model {
  for (j in 1:8){
    y[i] ~ normal(theta[i], sigma[i]);[8]
    theta_std[i] ~ normal(0, 1);
  }
}
```

One advantage of the original Stan code compared to SlicStan is the flexibility the user has to name all model parameters. In Stan, the auxiliary standard normal variables theta_std are named by the user, while in SlicStan, the names of parameters defined inside of a function are automatically generated, and might not correspond to the names of transformed parameters of interest. All parameter names are important, as they are part of the output of the sampling algorithm, which is shown to the user. Even though in this case the auxiliary parameters were introduced solely for performance reasons, inspecting their values in Stan's output can be useful for debugging purposes.

6 RELATED WORK

There exists a range of probabilistic programming languages and systems. Stan's syntax is inspired by that of BUGS [Gilks et al. 1994], which uses Gibbs sampling to perform inference. Other languages include Anglican [Wood et al. 2014], Church [Goodman et al. 2012] and Venture [Mansinghka et al. 2014], which focus on expressiveness of the language and range of supported models. They provide clean syntax and formalised semantics, but use less efficient, more general-purpose inference algorithms. The Infer.NET framework [Minka et al. 2014] uses an efficient inference algorithm called expectation propagation, but supports a limited range of models. Turing [Ge et al. 2018] allows different inference techniques to be used for different sub-parts of the model, but requires the user to explicitly specify which inference algorithms to use as well as their hyperparameters

More recently, there has been the introduction of *deep probabilistic programming*, in the form of Edward [Tran et al. 2018, 2016] and Pyro [Uber AI Labs 2017], which focus on using deep learning techniques for probabilistic programming. Edward and Pyro are built on top of the deep learning libraries TensorFlow [Abadi et al. 2016] and PyTorch [Paszke et al. 2017] respectively, and support a range of efficient inference algorithms. However, they lack the conciseness and formalism of some of the other systems, and it many cases require sophisticated understanding of inference.

[8] In the full version of Stan these statements can be "vectorised" for efficiency, e.g. `y ~ normal(theta,sigma);`

Other languages and systems include Hakaru [Narayanan et al. 2016], Figaro [Pfeffer 2009], Fun [Borgström et al. 2011], Greta [Golding et al. 2018] and many others.

The rest of this section addresses related work done mostly within the programming languages community, which focuses on the semantics (§§ 6.1), static analysis (§§ 6.2), and usability (§§ 6.3) of probabilistic programming languages. A more extensive overview of the connection between probabilistic programming and programming language research is given by Gordon et al. [2014b].

6.1 Formalisation of Probabilistic Programming Languages

There has been extensive work on the formalisation of probabilistic programming languages syntax and semantics. A widely accepted denotational semantics formalisation is that of Kozen [1981]. Other work includes a domain-theoretic semantics [Jones and Plotkin 1989], measure-theoretic semantics [Borgström et al. 2011; Ścibior et al. 2015; Toronto et al. 2015], operational semantics [Borgström et al. 2016a; Dal Lago and Zorzi 2012; Staton et al. 2016], and more recently, categorical formalisation for higher-order probabilistic programs [Heunen et al. 2017]. Most previous work specifies either a measure-theoretic denotational semantics, or a sampling-based operational semantics. Some work [Huang and Morrisett 2016; Hur et al. 2015; Staton et al. 2016] gives both denotational and operational semantics, and shows a correspondence between the two.

The density-based semantics we specify for Stan and SlicStan is inspired by the work of Hur et al. [2015], who give an operational sampling-based semantics to the imperative language Prob. Intuitively, the difference between the two styles of operational semantics is:

- Operational *density-based* semantics specifies how a program S is executed to evaluate the (unnormalised) posterior density $p^*(\boldsymbol{\theta} \mid \mathcal{D})$ at some specific point $\boldsymbol{\theta}$ of the parameter space.
- Operational *sampling-based* semantics specifies how a program S is executed to evaluate the (unnormalised) probability $p^*(\mathbf{t})$ of the program generating some specific trace of samples $\mathbf{t}$.

Refer to Appendix B.2 for examples and further discussion of the differences between density-based and sampling-based semantics.

6.2 Static Analysis for Probabilistic Programming Languages

Work on static analysis for probabilistic programs includes several papers that focus on improving efficiency of inference. R2 [Nori et al. 2014] applies a semantics-preserving transformation to the probabilistic program, and then uses a modified version of the Metropolis–Hastings algorithm that exploits the structure of the model. This results in more efficient sampling, which can be further improved by *slicing* the program to only contain parts relevant to estimating a target probability distribution [Hur et al. 2014]. Claret et al. [2013] present a new inference algorithm that is based on data-flow analysis. Hakaru [Narayanan et al. 2016] is a relatively new probabilistic programming language embedded in Haskell, which performs automatic and semantic-preserving transformations on the program, in order to calculate conditional distributions and perform exact inference by computer algebra. The PSI system [Gehr et al. 2016] analyses probabilistic programs using a symbolic domain, and outputs a simplified expression representing the posterior distribution. The Julia-embedded language Gen [Cusumano-Towner and Mansinghka 2018] uses type inference to automatically generate inference tactics for different sub-parts of the model. Similarly to Turing, the user then combines the generated tactics to build a model-specific inference algorithm.

With the exception of the work on slicing [Hur et al. 2014], which is shown to work with Church and Infer.NET, each of the above systems either uses its own probabilistic language or the method is applicable only to a restricted type of models (for example boolean probabilistic programs). SlicStan is different in that it uses information flow analysis and type inference in order to self-optimise to Stan — a scalable probabilistic programming language with a large user-base.

6.3 Usability of Probabilistic Programming Languages

This paper also relates to the line of work on usability of probabilistic programming languages. Gordon et al. [2014a] implement a schema-driven language, Tabular, which allows probabilistic programs to be written as annotated relational schemas. Fabular [Borgström et al. 2016b] extends this idea by incorporating syntax for hierarchical linear regression inspired by the lme4 package [Bates et al. 2014]. BayesDB [Mansinghka et al. 2015] introduces BQL (Bayesian Query Language), which can be used to answer statistical questions about data, through SQL-like queries. Other work includes visualisation of probabilistic programs, in the form of graphical models [Bishop et al. 2002; Gilks et al. 1994; Gorinova et al. 2016], and more data-driven approaches, such as synthesising programs from relational datasets [Chasins and Phothilimthana 2017; Nori et al. 2015].

7 CONCLUSION

Probabilistic inference is a challenging task. As a consequence, existing probabilistic languages are forced to trade off efficiency of inference for range of supported models and usability. For example, Stan, an increasingly popular probabilistic programming language, makes efficient scalable automatic inference possible, but sacrifices compositionality of the language.

This paper formalises the syntax of a core subset of Stan and gives its operational *density-based semantics*; it introduces a new, compositional probabilistic programming language, SlicStan; and it gives a semantic-preserving procedure for translating SlicStan to Stan. SlicStan adopts an *information-flow type system*, that captures the taxonomy classes of variables of the probabilistic model. The classes can be inferred to automatically optimise the program for probabilistic inference. To the best of our knowledge, this work is the first formal treatment of the Stan language.

We show that the use of static analysis and formal language treatment can facilitate efficient black-box probabilistic inference, and improve usability. Looking forward, it would be interesting to formalise the usage of pseudo-random generators inside of Stan. Variables in the **generated quantities** block can be generated using pseudo-random number generators. In other words, the user can explicitly compose Hamiltonian Monte Carlo with forward (ancestral) sampling to improve inference performance. SlicStan can be extended to automatically determine what the most efficient way to sample a variable is, which could significantly improve usability. Another interesting future direction would be to adapt the sampling-based semantics of Hur et al. [2015] to SlicStan and establish how the density-based semantics of this paper corresponds to it.

ACKNOWLEDGMENTS

We thank Bob Carpenter and the Stan team for insightful discussions, and the anonymous reviewers and George Papamakarios for useful comments. Maria Gorinova was supported by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. ACM, 147–160.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Douglas Bates, Martin Maechler, Ben Bolker, Steven Walker, et al. 2014. lme4: Linear mixed-effects models using Eigen and S4. *R package version* 1, 7 (2014), 1–23.

Michael Betancourt. 2014. Hamiltonian Monte Carlo and Stan. *Machine Learning Summer School (MLSS) lecture notes* (2014).

Michael Betancourt. 2017. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434* (2017).

Christopher M Bishop, David Spiegelhalter, and John Winn. 2002. VIBES: A variational inference engine for Bayesian networks. In *Advances in Neural Information Processing Systems*. 777–784.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016a. A Lambda-calculus Foundation for Universal Probabilistic Programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 33–46. https://doi.org/10.1145/2951913.2951942

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure Transformer Semantics for Bayesian Machine Learning. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 77–96. http://dl.acm.org/citation.cfm?id=1987211.1987216

Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo, Adam Ścibior, and Marcin Szymczak. 2016b. Fabular: Regression Formulas As Probabilistic Programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 271–283. https://doi.org/10.1145/2837614.2837653

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, Articles* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

Sarah Chasins and Phitchaya Mangpo Phothilimthana. 2017. Data-Driven Synthesis of Full Probabilistic Programs. In *International Conference on Computer Aided Verification*. Springer, 279–304.

Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 92–102.

Marco Cusumano-Towner and Vikash K. Mansinghka. 2018. A Design Proposal for Gen: Probabilistic Programming with Fast Custom Inference via Code Generation. *Workshop on Machine Learning and Programming Languages (MAPL)* (2018).

Ugo Dal Lago and Margherita Zorzi. 2012. Probabilistic operational semantics for the lambda calculus. *RAIRO-Theoretical Informatics and Applications* 46, 3 (2012), 413–450.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research)*, Amos Storkey and Fernando Perez-Cruz (Eds.), Vol. 84. PMLR, Playa Blanca, Lanzarote, Canary Islands, 1682–1690. http://proceedings.mlr.press/v84/ge18b.html

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian Data Analysis*. Chapman & Hall/CRC Press, London, third edition.

W R Gilks, A Thomas, and D. J. Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* 43 (1994), 169–178.

Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*. IEEE, 11–11.

Nick Golding, Tymoteusz Wołodźko, and Ben Marwick. 2018. Greta: Simple and Scalable Statistical Modelling in R. 2018. https://greta-dev.github.io/greta/index.html

Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Daniel Tarlow. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

Andrew D Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2014a. Tabular: a schema-driven probabilistic programming language. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 321–334.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014b. Probabilistic programming. In *Future of Software Engineering (FOSE 2014)*. 167–181.

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2018a. Probabilistic Programming with Densities in SlicStan: Efficient, Flexible and Deterministic. *arXiv preprint arXiv:1811.00890* (2018). https://arxiv.org/abs/1811.00890

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2018b. SlicStan: A Blockless Stan-like Language. *StanCon*. https://doi.org/10.5281/zenodo.1284348

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2018c. SlicStan: Improving Probabilistic Programming using Information Flow Analysis. *Probabilistic Programming Languages, Semantics, and Systems (PPS 2018)* (2018).

Maria I. Gorinova, Advait Sarkar, Alan F. Blackwell, and Don Syme. 2016. A Live, Multiple-Representation Probabilistic Programming Environment for Novices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 2533–2537. https://doi.org/10.1145/2858036.2858221

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. *arXiv preprint arXiv:1701.02547* (2017).

Matthew D Hoffman and Andrew Gelman. 2014. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.

Daniel Huang and Greg Morrisett. 2016. An application of computable distributions to the semantics of probabilistic programming languages. In *European Symposium on Programming Languages and Systems*. Springer, 337–363.

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2015. A Provably Correct Sampler for Probabilistic Programs. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India (LIPIcs)*, Prahladh Harsha and G. Ramalingam (Eds.), Vol. 45. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 475–488. https://doi.org/10.4230/LIPIcs.FSTTCS.2015.475

Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. 2014. Slicing probabilistic programs. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 133–144.

Claire Jones and Gordon D Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*. IEEE, 186–195.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *Journal of computer and system sciences* 22, 3 (1981), 328–350.

David Lunn, Christopher Jackson, Nicky Best, Andrew Thomas, and David Spieghalter. 2013. *The BUGS Book*. CRC Press.

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

Vikash Mansinghka, Richard Tibbetts, Jay Baxter, Pat Shafto, and Baxter Eaves. 2015. BayesDB: A probabilistic programming system for querying the probable implications of data. *arXiv preprint arXiv:1512.05006* (2015).

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Iain Murray. 2007. *Advances in Markov Chain Monte Carlo methods*. University of London, University College London (United Kingdom).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Functional and Logic Programming*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, Cham, 62–79.

Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* 2, 11 (2011).

Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs.. In *AAAI*. 2476–2482.

Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 208–217.

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).

Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137 (2009), 96.

Martyn Plummer et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, Vol. 124. Vienna, Austria.

Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

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

Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–176.

Geoffrey Smith. 2007. Principles of secure information flow analysis. *Malware Detection* (2007), 291–307.

Stan Development Team. 2017. Stan Modeling Language: User's Guide and Reference Manual. http://mc-stan.org.

Stan Development Team. 2018a. RStan: the R interface to Stan. http://mc-stan.org/ R package version 2.17.3.

Stan Development Team. 2018b. Stan GitHub Repositories. https://github.com/stan-dev

Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 525–534.

Marcin Szymczak. 2018. *Programming language semantics as a foundation for Bayesian inference*. Ph.D. Dissertation. The University of Edinburgh.

Sean J Taylor and Benjamin Letham. 2017. Forecasting at Scale. https://facebookincubator.github.io/prophet/.

Neil Toronto, Jay McCarthy, and David Van Horn. 2015. Running probabilistic programs backwards. In *European Symposium on Programming Languages and Systems*. Springer, 53–79.

Dustin Tran, Matthew D. Hoffman, Srinivas Vasudevan, Christopher Suter, Dave Moore, Alexey Radul, Matthew Johnson, and Rif A. Saurous. 2018. Edward2: Simple, Distributed, Accelerated. (2018). https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/edward2 To appear in Advances in Neural Information Processing Systems.

Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

Uber AI Labs. 2017. Pyro: A deep probabilistic programming language. http://pyro.ai/.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 1024–1032.