Probabilistic Programming with SlicStan

Maria I. Gorinova

Master of Science by Research

— Centre for Doctoral Training in Data Science —

School of Informatics

University of Edinburgh

August 16, 2017

Abstract

Probabilistic programming languages provide a concise and abstract way to specify probabilistic models, while hiding away the complicated underlying inference algorithm. However, those languages are often either not efficient enough to use in practice, or restrict the range of supported models and require understanding of how the compiled program is executed. Work in the programming languages community addresses this by attempting to use techniques such as program analysis and program abstraction, to improve the efficiency of the inference method used. However, such techniques have been restricted mainly within the community, and have not been applied to real-world probabilistic programming languages that have a large user-base.

This work seeks ways in which programming language and static analysis techniques can be used to improve the increasingly mature probabilistic language Stan. We design and implement SlicStan — a probabilistic programming language that uses methods from the programming languages community in a novel way, to allow for more abstract and flexible Stan models. We show with examples the functionality of SlicStan, and report compiler performance results to show that our method is practical. This work demonstrates that black-box efficient inference can be the result of joint efforts between programming language and machine learning researchers.

Acknowledgements

I would like to thank my supervisors **Andy Gordon** and **Charles Sutton** for conceiving the idea of such an interesting project, and for their invaluable advice, guidance, and support. This work was only possible as a result of many illuminating discussions and Charles's and Andy's ceaseless encouragement. The many meetings we had expanded my knowledge about both programming languages and machine learning, and emanated some exciting new ideas.

A very special thank you to **George Papamakarios** for his especially insightful comments and suggestions, proofreading this dissertation, and supporting the project throughout. The discussions we had taught me a lot about inference, inspired new ideas, and kept my motivation high.

Finally, thank you to my coursemates for all the engaging machine learning conversations, and especially to **Yanghao Wang**, whose suggestion contributed to a more complete analysis of how SlicStan's compiler scales.

This work was supported in part 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.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

August 16, 2017 (Maria I. Gorinova)

Contents

Chapter 1

Introduction

Machine learning has been growing faster and faster in the last couple of decades. We have witnessed huge advances in the areas of statistics, robotics, natural language processing, artificial intelligence, and many others, thanks to this increasingly growing field. In many cases such advances are due to the development of algorithms for *probabilistic inference*, and of programming languages that compile abstract model specifications for inference automatically [Wood et al., 2014, Lunn et al., 2000, Goodman et al., 2012, Minka et al., 2014, Gelman et al., 2015, Mansinghka et al., 2014]. Such *probabilistic programming languages* act as a high-level platform for users who are not statistics experts. They abstract away the details of the underlying inference algorithm, and provide a clean and concise syntax for probabilistic modelling.

Even though probabilistic programming has reached a point in which it can be applied to real-world problems, it still faces a lot of challenges. Implementing efficient inference algorithms that are applicable to a large number of different probabilistic models is hard. Languages that allow for a wide range of complex programs are forced to use less-efficient, more general-purpose inference methods. On the other hand, efficient inference algorithms usually make assumptions about the underlying model, which in turn imposes limits on the probabilistic language's expressiveness. Current languages face a trade-off between speed and accuracy of inference, and range of supported models and usability.

Stan [Gelman et al., 2015] is one such probabilistic programming language. Its nature is imperative, and programs compile to an efficient *Hamiltonian Monte Carlo* (HMC) inference algorithm [Neal et al., 2011]. The language is increasingly used for real-world scalable projects in statistics and data science (for example, Facebook's Prophet [Taylor and Letham, 2017]). Moreover, as programs are written in standalone files, and can be compiled using any of the available Stan interfaces,[1] the language is a convenient way to share and reproduce results of statistical models. Stan makes practical inference possible by (1) having syntax that allows separations between variables playing different roles in the HMC algorithm; (2) restricting the range of supported models to *fixed-sized* models that define a *differentiable* joint probability density function; and (3) adopting an adaptive version of HMC, which does not require algorithm parameters to be hand-tuned.

[1] To date those include command line, R, Python, MATLAB, Julia, Stata, and Mathematica interfaces.

Even though increasingly mature, and very powerful, Stan sacrifices some of its usability and flexibility to make automatic inference possible. In this dissertation, we motivate, design, and implement SlicStan — a probabilistic programming language that compiles to Stan. SlicStan uses static analysis and programming language techniques to allow models compiled for HMC inference to be more abstract and flexible. The rest of this chapter states the contributions of this work, also outlining the structure of the rest of the dissertation (§ 1.1), and finishes with a short review of related work (§ 1.2).

1.1 Contributions and Outline

SlicStan is novel in two ways:

(1) It uses *information flow analysis* and *type inference* to automatically determine the role that each program variable plays in HMC inference, and thus to eliminate the need to have *program blocks* as in Stan.

(2) It introduces user-defined functions that allow for new model parameters to be declared as local variables. Programs can then be translated to Stan, by statically unrolling calls to user-defined functions.

Below, we demonstrate SlicStan's functionality with an example. The Stan program (right) uses program blocks to specify the role of each variable and statements (**parameters**, **transformed parameters**, **model**), and contains no function calls. SlicStan (left) uses two function calls to the user-defined function my_normal to express the same model.[2] It does not contain any annotations as to what block a variable belongs to.

SlicStan

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

Stan

```
parameters {
  real y_raw;
  real x_raw;
}
transformed parameters {
  real y;
  real x;
  y = 3.0 * y_raw;
  x = exp(y/2) * x_raw;
}
model {
  y_raw ~ normal(0, 1);
  x_raw ~ normal(0, 1);
}
```

[2]The translated to Stan program is syntactically different to the original Stan program shown here, but it is semantically equivalent. We revisit this example in § 5.3, where we give more details.

A detailed explanation of Stan's syntax, as well as specifics about its underlying inference algorithm and the role of program blocks, is presented in the next chapter, Chapter 2.

Chapter 3 motivates and describes the idea behind SlicStan. As mentioned earlier, SlicStan translates to Stan. This process is described in full by Chapter 4, but in short, translation is done in three steps:

(1) *Typechecking:* the SlicStan program is checked for type errors, and the HMC role of each variable is deduced.

(2) *Elaboration:* calls to user-defined functions are statically unrolled.

(3) *Transformation:* variable declarations and statements are *shredded* into different blocks, based on the roles deduced by typechecking, to produce the final Stan translation of the original program.

Chapter 5 shows examples of programs that demonstrate the functionality of SlicStan, together with results reporting on the performance of the language's compiler.

1.2 Related Work

Today, there exists a range of probabilistic programming languages and systems. Stan's syntax, for example, is initially inspired by that of BUGS [Lunn et al., 2000], 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 all 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. On the other end of the spectrum, there is the Infer.NET framework [Minka et al., 2014], which uses the efficient variational inference algorithm Expectation Propagation, but allows a limited range of models to be specified. More recently, there was also the introduction of the Python library Edward [Tran et al., 2016], which supports a wide range of efficient inference algorithms and model criticism methods, but also has a limited range of supported models and lacks the conciseness and formalism of some of the other systems.

The rest of this section addresses mostly related work done within the programming languages community. A more extensive overview of the connection between probabilistic programming and programming language research si given by Gordon et al. [2014b].

1.2.1 Static Analysis for Probabilistic Programming Languages

Our work aligns most closely with the body of work on static analysis for probabilistic programs.

Such work includes several papers that attempt to improve efficiency of inference. For example, 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 a more efficient Metropolis–Hastings sampling, which can be further improved by *slicing* the probabilistic program to only contain parts that a relevant to estimating some target probability distribution [Hur et al., 2014]. R2 designs and uses its own probabilistic language, however slicing is shown to work with other languages, in particular Church and Infer.NET.

Another related work is that of Claret et al. [2013], who present a new inference algorithm that is based on data-flow analysis. Their approach is applicable to *boolean* probabilistic programs — programs where variables can only be of a boolean type.

The PSI system [Gehr et al., 2016] analyses probabilistic programs using a symbolic domain, and outputs a simplified expression representing the posterior distribution. PSI uses its own modelling language, and the method has not been applied to other already existing languages.

1.2.2 Usability of Probabilistic Programming Languages

Another group of related work is that done in terms of usability for probabilistic programming languages, and focuses on making inference more accessible.

Contributions in this area include introducing visual aids for the programmer in the form of graphical models expressing the underlying probabilistic program. WinBUGS [Lunn et al., 2000] and Vibes [Bishop et al., 2002] allow a model to be expressed either in a textual language (BUGS language/XML script) or pictorially, via a graphical user interface. Gorinova et al. [2016] create a development environment, which shows live, edit-triggered visualisations of an Infer.NET probabilistic model written by the user, but does not allow models to be specified graphically.

In other work, Gordon et al. [2014a] implement a schema-driven probabilistic programming language, Tabular, which brings the power of probabilistic modelling to spreadsheets, by allowing programs to be written as annotated relational schemas. Fabular [Borgström et al., 2016a] extends this idea by incorporating special syntax for hierarchical linear regression inspired by R and its lme4 package [Bates et al., 2014]. Finally, BayesDB [Mansinghka et al., 2015] introduces BQL — a Bayesian Query Language. Users can write SQL-like queries to answer their statistical questions about the data, and write probabilistic models in a minimal meta-modelling language.

Nori et al. [2015], and more recently Chasins and Phothilimthana [2017], have a more data-driven approach to usability of probabilistic programs, that is they synthesise programs from relational datasets. Nori et al. [2015] require the user to write a "sketch" program, which captures some of what the intended probabilistic model is, but contains "holes". These holes are substituted for actual code, after analysis of the data. Chasins and Phothilimthana [2017] extend this approach by automating the "sketching" of the program, although they must restrict the format of the input dataset to implement this idea.

1.2.3 Formalisation of Probabilistic Programming Languages

There has been extensive work on the formalisation of probabilistic programming languages syntax and semantics. For example, a widely accepted denotational semantics formalisation is that of Kozen [1981]. Other work includes formalisation of domain-theoretic semantics [Jones and Plotkin, 1989], measure-theoretic semantics [Borgström et al., 2011], operational semantics [Borgström et al., 2016b], and more recently, categorical formalisation for higher-order probabilistic programs [Heunen et al., 2017]. In this dissertation, we define the semantics of SlicStan in terms of Stan. Giving the language formal semantics is out of the scope of this dissertation, however we consider it an interesting direction for future work.

Chapter 2

Probabilistic Programming with Stan

This chapter outlines the theoretical and practical background of SlicStan. Firstly, § 2.1 briefly describes some of the basic syntax of the Stan programming language. To understand how to optimise their probabilistic programs, Stan users need to also have some insight on the way inference is implemented by Stan's compiler. A high-level introduction to the default inference algorithm, NUTS [Hoffman and Gelman, 2014], is presented in § 2.2, while § 2.3 highlights with examples how a Stan program can be optimised. As part of the main work of this dissertation, we also formally define the syntax of a subset of the Stan language, and state what a well-formed Stan program is (§§ 4.3.1).

2.1 Modelling Language

Stan [Gelman et al., 2015] is a probabilistic programming language, whose syntax is similar to that of BUGS [Lunn et al., 2000], and aims to be close to the model specification conventions used in publications in the statistics community. This section outlines the basic syntax and structure of a Stan program, and gives an example demonstrating most of the relevant to this dissertation language structures. Some syntactic constructs and functionality (such as constraint data types, random number generation, penalized maximum likelihood estimation, and others) have been omitted. A full language specification can be found in the official reference manual [Stan Development Team, 2017].

2.1.1 Overview

Stan is imperative and compiles to C++ code. It consists of program blocks (§§ 2.1.2), which contain sequences of variable declarations and statements. Statements themselves involve expressions, which can be constants, variables, arithmetic expressions, array expressions, function calls, and others.

The *meaning* of a Stan program is the joint probability density function $P(x_1, \ldots, x_n)$ it defines (up to a constant), where $x_1, \ldots, x_n$ are the parameters of the model. The

execution of a Stan program consists of drawing samples from this joint density.

A full example of a Stan program is shown in Figure 2.1.

Data Types and Declarations

All variables in Stan must be declared before use, and their types must be specified. This is done in a similar to C++ way. For example, the following will declare a single floating point variable x:

```
real x;
```

The language supports the following basic types: **int**, **real**, **vector**, **row_vector**, **matrix** and an array of any type can be created. Linear algebra objects (vector, row vector and matrix) have real elements. The size of vectors, row vectors, matrices and arrays must be specified when declaring a variable of such type:

```
real x[10];             // real array of size 10
vector[N] y;            // vector of size N
matrix[N, 4] m[3];      // array of 3 matrices of shape (N,4)
```

Statements

As mentioned in the beginning of this section, Stan is an imperative language. Thus, unlike BUGS where the program describes a Bayesian Network in a declarative fashion, statements are executed in the order they are written, and define the target log probability density function. Stan shares statements similar to those of other imperative languages — assignments, **for** and **while** loops, and **if** statements:

```
for (n in 1:N) {
  if (x[n] > 0) y[n] = x[n];
  else y[n] = 0;
}
```

In addition, Stan also supports a *sampling* statement:

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

Despite its name, a sampling statement in Stan does not actually perform any sampling when executed. The code above means that the variable y has a standard normal distribution in the probabilistic model we are defining. As the semantics of a Stan program is the log probability function it defines, a sampling statement is essentially an increment of this target log density:

$$log\, P_{i+1}(y, x_1, \ldots, x_n) = \log\, P_i(y, x_1, \ldots, x_n) + \log\, \mathcal{N}(y \,|\, 0, 1)$$

where $\log P_i(y, x_1, \ldots, x_n)$ is the accumulated log density at the i^{th} step of the execution. Stan also allows this increment to be done manually, by accessing a special **target** variable:

```
target += normal_lpdf(y | 0, 1);
```

2.1.2 Program Blocks

A full Stan program consists of 7 blocks, all of which are optional. The blocks (if present) must be in the following order, and are used to declare a particular set of variables and statements:

- **functions**: contains the definitions of user-defined functions (§§ 2.1.3).
- **data**: contains declarations of the variables, whose value will be fed externally before inference is performed.
- **transformed data**: contains declarations of known constants and allows for preprocessing of the data.
- **parameters**: contains declarations of the parameters of the model.
- **transformed parameters**: contains declarations and statements defining transformations of the data and parameters, that are used to define the target density.
- **model**: contains local variable declarations and statements defining the target log probability density function.
- **generated quantities**: contains declarations and statements that do not affect the sampling process, and are used for post-processing, or predictions for unseen data.

Figure 2.1 shows an example of a full Stan program. In this example, we are interested in inferring the mean μ_y and variance σ_y^2 of an array of independent and identically distributed variables $\mathbf{y}$. The program can be read as follows:

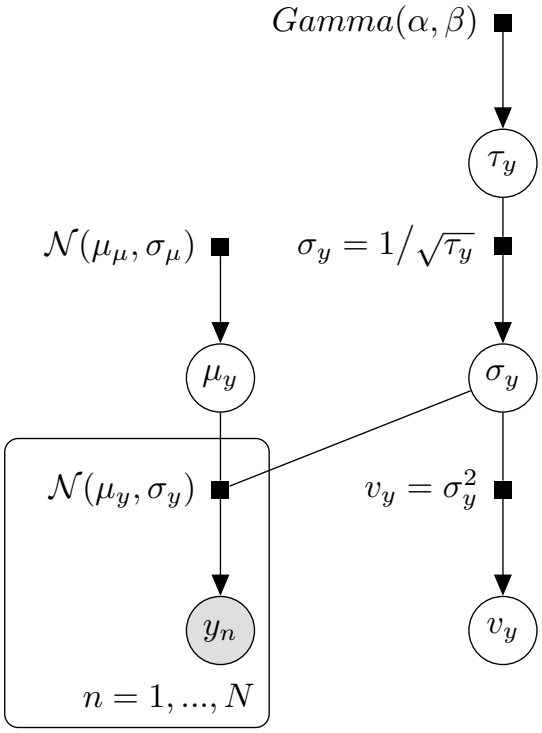
(1) In the **model** block, we specify that $\mathbf{y}$ comes from a normal distribution with mean μ_y and standard deviation σ_y, and give prior distributions to the parameters (declared in **parameters**) μ_y and τ_y.

(2) In the **transformed parameters** block, we define $\mathbf{y}$'s standard deviation σ_y in terms of $\mathbf{y}$'s precision τ_y.

(3) Similarly, we define the variance in terms of the transformed parameter σ_y in **generated quantities**.

(4) Finally, we declare the data items that will be specified before inference in the **data** block, and define the constants α and β in the **transformed data** block.

Declaring some of the variables in another block could also result in a valid Stan program (e.g. α, β, and σ^2 can be all defined in the **transformed parameters** block), however this might impact performance. More details about what block a variable should be defined in, in order to optimise a Stan program, is given in § 2.3.




```
data {
  int N;
  real y[N];
  real mu_mu;
  real sigma_mu;
}
transformed data {
  real alpha = 0.1;
  real beta = 0.1;
}
parameters {
  real mu_y;
  real tau_y;
}
transformed parameters {
  real sigma_y;
  sigma_y = pow(tau_y, -0.5);
}
model {
  tau_y ~ gamma(alpha, beta);
  mu_y ~ normal(mu_mu, sigma_mu);
  y ~ normal(mu_y, sigma_y);
}
generated quantities {
  real variance_y;
  variance_y = sigma_y * sigma_y;
}
```

Figure 2.1: An example of a probabilistic model written in Stan (on the right), and its corresponding factor graph (on the left). Example adapted from Stan's Reference Manual [Stan Development Team, 2017, p. 101].

2.1.3 User-defined functions

User-defined functions are supported, and they can be defined in the **functions** block. Functions must carry a name with a special suffix in the cases where the function accesses the accumulated log probability (_lp), or implements a probability function (_lpdf, _lpmf), or a random number generator (_rng).

Due to Stan's requirement that all parameters of the model should be declared in the **parameters** block, those user-defined functions are constrained, in the sense that they cannot define new parameters. In some cases, allowing parameters to be created within functions could lead to a more compact and easy to read program. Examples of such desirable functions, which cannot be implemented in the current version of Stan, are given in § 2.3.

2.2 Inference in Stan

The meaning of a Stan program is the joint probability density function it defines (up to a constant), or in other words, $P(\mathcal{D}, \boldsymbol{\theta})$, where $\mathcal{D}$ is the data and $\boldsymbol{\theta}$ is the parameters of the model. For example, the model specified in Figure 2.1 defines the density:

$$\begin{aligned} P(\mathbf{y}, \mu_y, \tau_y) &= Gamma(\tau_y \mid \alpha, \beta) \times \mathcal{N}(\mu_y \mid \mu_\mu, \sigma_\mu) \times \mathcal{N}(\mathbf{y} \mid \mu_y, \sigma_y) \\ &= Gamma(\tau_y \mid 0.1, 0.1) \times \mathcal{N}(\mu_y \mid \mu_\mu, \sigma_\mu) \times \mathcal{N}\left(\mathbf{y} \mid \mu_y, \frac{1}{\sqrt{\tau_y}}\right) \end{aligned}$$

with μ_μ and σ_μ being given unmodelled data, and $Gamma(x \mid \alpha, \beta)$ and $\mathcal{N}(x \mid \mu, \sigma)$ being the gamma and the Gaussian distribution respectively.

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*. In the example above, that is finding the posterior $P(\mu_y, \tau_y \mid \mathbf{y})$.

Currently, Stan supports the following ways to perform inference:

- Asymptotically exact Bayesian inference using Hamiltonian Monte Carlo (HMC) [Neal et al., 2011], a Markov Chain Monte Carlo (MCMC) method, or its "adaptive path lengths" extension — the No-U-Turn Sampler (NUTS) [Hoffman and Gelman, 2014].
- Approximate inference using Automatic Differentiation Variational Inference (ADVI) [Kucukelbir et al., 2016].

This dissertation is focused on improving Stan's syntax, in order to be able to compile models to Stan code, which results in a more efficient exact inference using the default algorithm — NUTS. The rest of this section, therefore, gives a brief introduction to Markov Chain Monte Carlo sampling (§§ 2.2.1), as well as the particular MCMC methods that Stan uses — Hamiltonian Monte Carlo and the No-U-Turn Sampler (§§ 2.2.2). Finally, §§ 2.2.3 outlines the way a Stan model is run to perform inference.

2.2.1 Markov Chain Monte Carlo (MCMC)

Monte Carlo methods aim to sample from a *target* probability distribution $P(\mathbf{x})$. Having obtained a collection of samples $\{\mathbf{x}_n\}_{n=1}^N$, we can then approximate expectations of any function $\phi(\mathbf{x})$ under that distribution, by taking:

$$\begin{aligned} \langle \phi(\mathbf{x}) \rangle_{P(\mathbf{x})} &= \int \phi(\mathbf{x})\, P(\mathbf{x})\, d\mathbf{x} \\ &\approx \frac{1}{N} \sum_{n=1}^{N} \phi(\mathbf{x}_n) \end{aligned} \tag{2.1}$$

Inference implemented using Monte Carlo methods consists of obtaining a representative collection of samples of the posterior distribution $P(\boldsymbol{\theta} \mid \mathcal{D})$, and treating this collection as an approximation of the distribution itself.

The idea behind Markov Chain Monte Carlo methods (introduced by Metropolis et al. [1953], and reviewed by Neal [1993] and Murray [2007]) is to obtain those samples by transitioning on a Markov chain, which is designed such that in the limit of infinitely many transitions, the samples obtained will be from the target probability distribution, or $P(\boldsymbol{\theta} \mid \mathcal{D})$ in the case of inference. In other words, suppose we define the probability of obtaining a collection of samples $\{\boldsymbol{\theta}_n\}_{n=1}^N$ to be the following Markov chain:

$$Q(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N) = Q(\boldsymbol{\theta}_1) \prod_{n=2}^{N} Q(\boldsymbol{\theta}_n \mid \boldsymbol{\theta}_{n-1})$$

Suppose also, that the stationary distribution $Q_\infty(\boldsymbol{\theta})$ of the Markov chain exists, and that we have proven that $Q_\infty(\boldsymbol{\theta}) = P(\boldsymbol{\theta} \mid \mathcal{D})$. Now, due to the chain converging towards its stationary distribution, drawing samples from $P(\boldsymbol{\theta} \mid \mathcal{D})$ by forward-sampling from the Markov chain $Q(\boldsymbol{\theta}_n \mid \boldsymbol{\theta}_{n-1})$, will result in asymptotically exact inference. In other words, in the presence of infinitely many samples, MCMC inference will be exact.

This raises the question of how many samples we need to obtain in practice, in order to approximate the posterior well. The number varies hugely, depending on the precise MCMC method used, and the nature of the distribution. For example, Metropolis–Hastings [Hastings, 1970], which is one such MCMC method used primarily for multivariate, complex distributions, will obtain a potential sample point $\boldsymbol{\theta}^*$, by taking a (random) step near the current sample point $\boldsymbol{\theta}_n$. It will then accept or reject $\boldsymbol{\theta}^*$ based on some acceptance ratio, and take $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}^*$ if $\boldsymbol{\theta}^*$ is accepted, and $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n$ otherwise. If the stepsize of this random walk is small, then it will take us a long time to "explore" the distribution, and obtain an effective collection of samples. Furthermore, it can be problematic to explore certain distributions, such as multimodal distributions, as we might never "step far enough" to reach a region close to another mode. On the other hand, if the stepsize is large, a random step is more likely to result in a low-probability proposal sample point, which in turn is likely to be rejected. The acceptance rate will drop, and it will, once again, take us a large number of iterations to obtain an effective collection of samples.

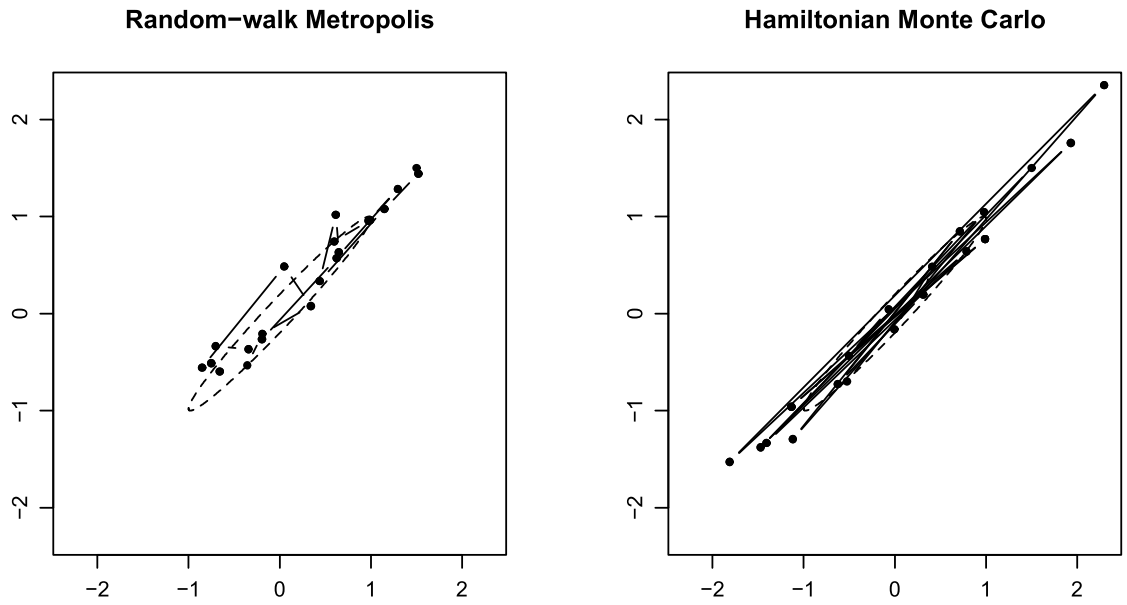


Figure 2.2: Twenty iterations of the random-walk Metropolis–Hastings method, and of the Hamiltonian Monte Carlo method. The target density is a 2D Gaussian distribution with marginal standard deviations of one and a high correlation of 0.98. Metropolis–Hastings takes small, target distribution-independent steps, while Hamiltonian Monte Carlo takes larger steps in the directions of the gradient, resulting in a better density exploration. Figure taken from Neal et al. [2011].

2.2.2 Hamiltonian Monte Carlo (HMC) and the No-U-Turn Sampler (NUTS)

As previously stated in the beginning of § 2.2, Stan uses two MCMC sampling methods — Hamiltonian Monte Carlo, and an adaptive tuning version of it: the No-U-Turn Sampler. Those two are outlined in turn below.

Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) [Neal et al., 2011] is an MCMC method, which attempts to reduce the problematic random walk behaviour described in the previous section, by making use of information about the target distribution's gradient. HMC augments the probability space with momentum variables, to sample by performing Hamiltonian dynamics simulation. This method makes it possible to propose a sample potentially distant from the current one, yet having a high acceptance probability. Figure 2.2 compares Metropolis–Hastings and Hamiltonian Monte Carlo on a single example, illustrating the importance of reducing random walk behaviour when using MCMC methods.

An intuition about the way HMC works is the following: Suppose that $P(\mathbf{x})$ is a differentiable probability density function. Imagine the negative log of $P(\mathbf{x})$ as a surface, and a single sample from the distribution, $\mathbf{x}$ being a frictionless puck placed on the surface. The *potential energy* of that puck, $E(\mathbf{x})$, is the height of the surface up to an additive

constant:

$$E(\mathbf{x}) = -\log P(\mathbf{x}) - \log Z$$

The *kinetic energy* of the puck is a function of its *momentum* $\mathbf{p}$:

$$K(\mathbf{p}) = \frac{1}{2m}\|\mathbf{p}\|^2$$

where m is the mass of the puck, and $\|\mathbf{p}\|$ is the magnitude of $\mathbf{p}$.

Given an initial "push" in some direction, the puck slides on the surface, with:

- its potential energy $E(\mathbf{x})$ increasing and kinetic energy $K(\mathbf{p})$ decreasing, when going up a slope, and
- its potential energy $E(\mathbf{x})$ decreasing and kinetic energy $K(\mathbf{p})$ increasing, when going down a slope.

In a perfectly simulated system with a frictionless puck, the *Hamiltonian*, defined as a function of the position $\mathbf{x}$ and the momentum $\mathbf{p}$, stays constant:

$$H(\mathbf{x}, \mathbf{p}) = E(\mathbf{x}) + K(\mathbf{p})$$

Hamiltonian Monte Carlo samples from the joint density $P_H(\mathbf{x}, \mathbf{p}) = \frac{e^{-H(\mathbf{x},\mathbf{p})}}{Z_H}$, by simulating the "sliding puck" physical system. Marginalising over $\mathbf{p}$ gives us:

$$\begin{aligned}
\int_{-\infty}^{+\infty} P_H(\mathbf{x}, \mathbf{p})\,\mathrm{d}\mathbf{p} &= \int_{-\infty}^{+\infty} \frac{1}{Z_H} e^{-E(\mathbf{x})} e^{-K(\mathbf{p})}\,\mathrm{d}\mathbf{p} \\
&= \frac{1}{Z} e^{-E(\mathbf{x})} \int_{-\infty}^{+\infty} \frac{Z}{Z_H} e^{-K(\mathbf{p})}\,\mathrm{d}\mathbf{p} \\
&= P(\mathbf{x})
\end{aligned}$$

Thus, assuming the simulation is perfect, the location of that puck at some point in time will be a sample from the distribution of interest.

In very simple terms, HMC then works by simply picking an initial sample $\mathbf{x}_0$, and then adding a new sample as follows:

(1) Place the puck at the position of the current sample $\mathbf{x}_n$.

(2) Choose a Gaussian initial momentum $\mathbf{p}$ to give to the puck.

(3) Simulate the physical system for some time τ.

(4) Set the candidate sample $\mathbf{x}^*$ to the position of the puck at the end of the simulation.

(5) Accept / reject the candidate sample based on a Metropolis acceptance ratio.[1]

[1] This step would not be needed if the simulation of the system were numerically perfect. In practice, we need to discretise time, and move in a straight line between individual time-steps. Therefore a simulation of the physical system is only approximate.

Hamiltonian Monte Carlo has proven its success in practise, including when used with Hierarchical models [Betancourt and Girolami, 2015], which are of particular importance to Stan. Not surprisingly, Stan's development team chooses HMC as the underlying inference algorithm. Implementing the algorithm, however, involves tuning several hyperparameters — a non-trivial, often difficult and model-dependent step. The default for Stan No-U-Turn Sampler automatically sets some of those parameters, and therefore allows for an easier use of Stan for the programmer.

The No-U-Turn Sampler

To implement Hamiltonian Monte Carlo that samples from a distribution $P(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{Z}$, one needs to:

- Know how to compute the gradient $\frac{\partial E}{\partial \mathbf{x}}\Big|_{\mathbf{x}=\mathbf{x}^*}$ at any point $\mathbf{x}^*$. Stan uses automatic differentiation to do that.
- Choose the *mass* m that defines the kinetic energy $K(\mathbf{p})$. In the general case, the mass is a matrix Σ, so $K(\mathbf{p}) = \frac{1}{2}\mathbf{p}^T\Sigma^{-1}\mathbf{p}$ and $\mathbf{p} \sim \mathcal{N}(\mathbf{0}, \Sigma)$.
- Choose a small time-interval ϵ — the time discretisation.[2]
- Choose the number of time-steps, L, to be taken to complete each simulation.[3]

The quantities Σ, ϵ and L are the parameters of the algorithm. As shown by Neal et al. [2011] and Hoffman and Gelman [2014], HMC efficiency is highly sensitive to those parameters. For example, a small L will result in a candidate sample close to the current one, increasing sample correlation. A large L, on the other hand, will result in slower sampling, as the algorithm will spend longer simulating a trajectory for each sample. Moreover, the simulation might result in a *U-turn* of the fictional puck, in which case continuing it is unnecessary, and it might even lead to a complete re-tracing of the steps, sending the puck back to where it started from.

The No-U-Turn Sampler (NUTS) [Hoffman and Gelman, 2014] is an HMC algorithm that chooses the path length L on the fly, by creating a trajectory of candidate points, and sampling from those points, taking care to preserve the necessary properties of the Markov chain for MCMC sampling to work. NUTS also tunes the step size ϵ to match some target acceptance-rate δ (the default value is $\delta = 0.8$). Finally, the mass matrix Σ is automatically set during NUTS's *warm-up phase* — a sampling phase, obtained samples from which are later discarded — to account for possible correlations between parameters.

Thanks to NUTS, Stan is able to perform efficient inference on hierarchical models, without any hand-tuning needed to be done by the programmer.

[2] Also called "step size".
[3] Also called "path length".

Algorithm 1: Vanilla HMC applied to a Stan program.[1]

Arguments: $\mathbf{d}$, the data // Read **data** block variables
Returns: (M, T, Q), a set of samples

$\mathbf{t_d} = td(\mathbf{d})$ // Compute **transformed data** variables
for all $s \in 1 : S$ **do** // Loop over samples
  $\mathbf{p} \sim \mathcal{N}(0, \Sigma)$ // Sample initial momentum
  $H = \frac{\mathbf{p^T p}}{2} + E(\boldsymbol{\theta}_n)$ // Evaluate the Hamiltonian

  $\boldsymbol{\theta}^* = \boldsymbol{\theta}_n$ // Start from current sample
  for all $\tau \in 1 : L$ **do** // Take L **leapfrog** steps
    $\mathbf{p} = \mathbf{p} - \frac{\epsilon}{2} \times \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*}$ // Make a half-step in $\mathbf{p}$
    $\boldsymbol{\theta}^* = \boldsymbol{\theta}^* + \epsilon \times \Sigma^{-1}\mathbf{p}$ // Update the **parameters**
    $\mathbf{p} = \mathbf{p} - \frac{\epsilon}{2} \times \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*}$ // Find new gradient (involves
    // computing the **transformed parameters**,
    // and executing the **model** block),
    // and make a half-step in $\mathbf{p}$

  $H^* = \frac{\mathbf{p^T p}}{2} + E(\boldsymbol{\theta}^*)$ // Re-evaluate the Hamiltonian
  $\delta H = H^* - H$
  if $(\delta H < 0) \vee (rand() < e^{-\delta H})$ **then** // Decide whether to accept or reject
    accept $= true$ // the candidate sample $\boldsymbol{\theta}^*$
  else accept $= false$

  if accept **then** $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}^*$
  else $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n$

  $M = M \cup \{\boldsymbol{\theta}_{n+1}\}$ // Add $\boldsymbol{\theta}_{n+1}$ to the set of sampled parameters
  $T = T \cup \{tp(\boldsymbol{\theta}_{n+1})\}$ // Add transformed parameters to the samples
  $Q = Q \cup \{g(\boldsymbol{\theta}_{n+1})\}$ // Compute and include **generated quantities**

return (M, T, Q)

[1] The pseudo-code was adapted from *"Information Theory, Inference and Learning Algorithms"* [MacKay, 2003, p. 388] for illustrative purposes. It excludes details of the actual algorithm used by Stan, such as transformations of constrained variables, gradient computation, parameter tuning, and others. The full source-code of Stan is available online at https://github.com/stan-dev/stan, and a sketch of the way a compiled program is executed is given in the manual [Stan Development Team, 2017, p. 115].

2.2.3 Execution of a Stan Program

Going back to Stan's syntax, this subsection outlines how code written in Stan transforms to an efficient Hamiltonian Monte Carlo algorithm. Algorithm 1 shows a vanilla HMC pseudo-code, with information about executing lines of code from different Stan blocks, in order to sample from the desired probability density.

The first step is loading the data, as declared in the **data** block, into memory. Then the statements in the **transformed data** block are executed to calculate the transformed data variables. For each sample, the algorithm executes a certain number of *leapfrog* steps — the simulation of the sliding puck physical system. Each leapfrog step is a small move of the puck in the direction of its momentum, and a respective small change of its momentum in the direction of its gradient. The target density is defined in terms of the data, transform data, parameters, and transformed parameters. Re-evaluating the gradient of this target density requires executing the **transformed parameters** block to re-calculate the transformed parameters (as they are defined in terms of the now updated parameters). After the simulation is complete, a Metropolis acceptance ratio is computed to decide if the sample should be accepted or rejected. Then the **generated quantities** block is executed for every sample.

Table 2.1 summarises how frequently statements in different block are executed. It is clear that equivalent Stan programs, in which variables have been declared and defined in different blocks, can have huge differences in performance — the data is transformed only once (per execution), while the algorithm spends most of its time in the inner, leapfrog loop, where the **transformed parameters** and **model** blocks are executed. It is therefore wasteful to define a variable that is a transformation of the data, or a generated quantity, in the **transformed parameters** block.

Block	Execution
data	—
transformed data	Per chain
parameters	—
transformed parameters	Per leapfrog
model	Per leapfrog
generated quantities	Per sample

Table 2.1: Execution of program blocks in Stan. Adapted from [Betancourt, 2014].

We address this and other issues in the development of SlicStan.

2.3 Optimising Stan Code

The specifics of the underlying inference algorithm of Stan, Hamiltonian Monte Carlo sampling, leave a lot of room for optimisations of the source code of a probabilistic model, that will lead to more efficient sampling. This section outlines a few code optimisation

techniques that are of importance to this dissertation. A more detailed list of optimisation techniques is presented in ch. 26 of Stan's Manual [Stan Development Team, 2017].

2.3.1 Program Blocks

First of all, as explained in §§ 2.2.3, the way variables' definitions are sliced between blocks has an impact on the performance of the compiled Stan program. Therefore, it is important to ensure that each variable is defined in the optimal block. In short:

- Constants and transformations of data (variables defined as data or transformed data), should be placed in the **transformed data** block.
- Variables that are defined in terms of data and parameter variables, but do not take part in the probability density's definition (i.e. for a variable `q` there are no statements such as `q ~ foo(...)` or `x ~ foo(q, ...)`), should be defined in the **generated quantities** block.
- Only variables that cannot be defined as transformed data or generated quantities should be defined in the **transformed parameters** block.

2.3.2 Reparameterisation

When the mass matrix Σ is fixed, as it is the case for the current version of Stan, Hamiltonian Monte Carlo uses a first-order approximation to the posterior. This means that if the posterior has a difficult geometry, which cannot be approximated well without taking into account its curvature, Stan's sampler may become slow, and even not sample from the correct density. Such geometries can occur, for example, when the posterior has strong non-linear correlations, as in the example described below. For such situations, the Stan Development Team [2017] suggests attempting to reparameterise the model, so that the HMC parameters' space becomes flatter.[4] The rest of this section demonstrates the "difficult geometries" problem with an example, and presents an efficient reparameterisation, which makes sampling more efficient.

Neal's Funnel

One pathological example is the *Neal's Funnel* example, which was chosen by Neal [2003] to demonstrate the difficulties Metropolis–Hastings runs into when sampling from a distribution with strong non-linear correlations. The probability density this model defines is over a 9-dimensional vector $\mathbf{x}$ and a variable y, such that:

$$y \sim \mathcal{N}(0, 3)$$
$$\mathbf{x} \sim \mathcal{N}(\mathbf{0}, e^{\frac{y}{2}}\mathbb{I})^{5}$$

[4] Alternatively, Riemannian Manifold Hamiltonian Monte Carlo (RMHMC) [Girolami et al., 2009] overcomes the difficult geometries problem by making the mass matrix position-dependent. However, RMHMC is very expensive computationally, therefore harder to use in practise.

[5] Here $\mathbf{0}$ is the zero vector, and $\mathbb{I}$ is the identity matrix.

The density has the form of a 10-dimensional funnel (thus the name *"Neal's Funnel"*), with a very sharp neck, as shown in Figure 2.3c. Due to its fixed mass matrix and first-order posterior approximation, Hamiltonian Monte Carlo has trouble sampling from this distribution. Figure 2.3a demonstrates this by plotting $24,000$ samples drawn using Stan's sampler, and using the following *straightforward* way to specify the model in Stan:

```
parameters {
  real y;
  vector[9] x;
}
model {
  y ~ normal(0, 3);
  x ~ normal(0, exp(y/2));
}
```

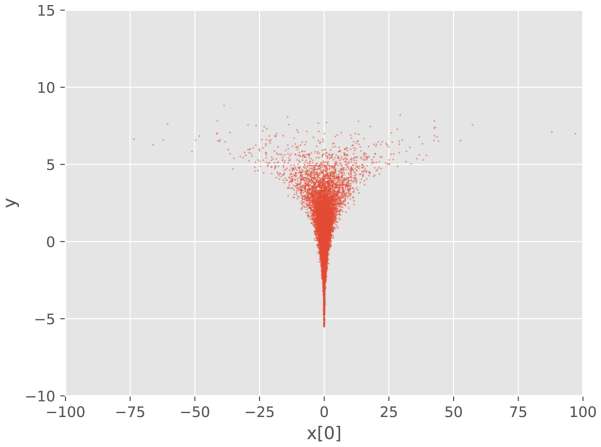
It can be seen that the sampler has trouble obtaining samples from the neck of the funnel. This is because both the step-size ϵ, and the mass matrix Σ are set during warm-up, and stay constant when drawing actual samples. This works well in cases of linear correlations between parameters, where the curvature of the probability density is constant, but $\mathbf{x}$ and y here are non-linearly correlated. Choosing the step-size and mass matrix to work well in the body, means that it will be difficult to explore the neck, while optimising those parameters for the neck, will result in an inefficient exploration of the body.

A more *efficient* model can be specified in Stan, by reparameterisation — samples can be drawn from independent standard normal variables, and $\mathbf{x}$ and y obtained from those:

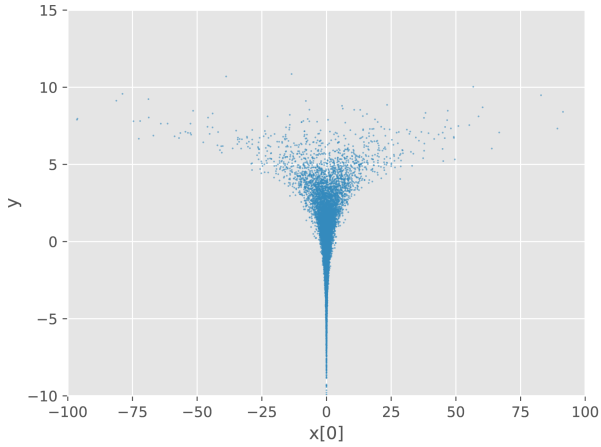
```
parameters {
  real y_raw;
  vector[9] x_raw;
}
transformed parameters {
  real y;
  vector[9] x;
  y = 3.0 * y_raw;
  x = exp(y/2) * x_raw;
}
model {
  y_raw ~ normal(0, 1); // implies y ~ normal(0, 3)
  x_raw ~ normal(0, 1); // implies x ~ normal(0, exp(y/2))
}
```

Even though longer and less readable, this model performs much better than the straightforward one. Figure 2.3b shows $24,000$ samples drawn using Stan's sampler, but this time using the more efficient model specification above.

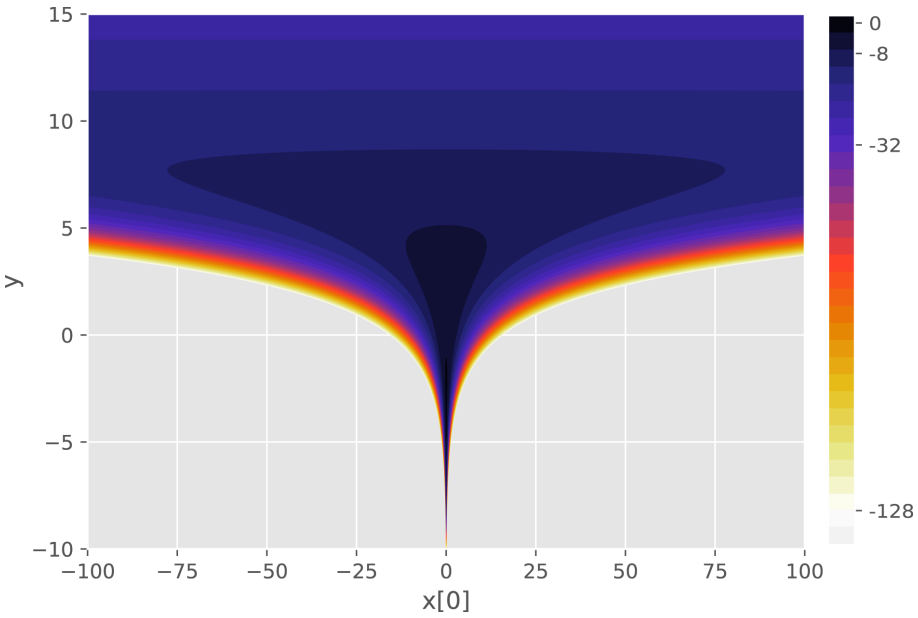
Neal's Funnel is an extreme, but typical example of the correlations that priors in hierarchical models could have. Neal [2003] suggests that x_1 to x_9 can be random effects for 9 subjects, and y — the log of the variance of those effects. If we don't have enough



(a) 24,000 samples obtained from 6 runs of Stan's sampler (default settings) for the *non-efficient* form of the Neal's Funnel model.



(b) 24,000 samples obtained from 6 runs of Stan's sampler (default settings) for the *efficient* form of the Neal's Funnel model.



(c) The actual log density of Neal's Funnel example. Darkest regions are of high density, with log density greater than -8, while grey regions have log density of less than -128.

Figure 2.3

informative data, the posterior will be of similar shape to that of the prior, i.e. it will be a 10 dimensional funnel with a very sharp neck. At the same time, it is sometimes difficult to detect problems with sampling from models with a problematic posterior. Problems such as that of Neal's Funnel could easily be left undetected in a real-world hierarchical model. Thus, such reparameterisations occur often, sometimes many times per model.

In practice, there exist many forms of reparameterisation that could lead to improved efficiency. Some examples are shown in Chapter 5 to demonstrate the usability of SlicStan. A large collection of examples is also given in Stan's Reference Manual [Stan Development Team, 2017].

2.3.3 Vectorisation

Earlier in this chapter (§§ 2.2.3) we outlined the way a Stan program is executed. Going back to Algorithm 1's pseudo-code, we see that the gradient of the target probability density at the current sample point needs to be evaluated as often as at every leapfrog iteration. This computation is both frequent and expensive, and Stan spends the majority of its execution time to implement it. Therefore, reducing the time required for the gradient to be evaluated can lead to drastic improvements in the time taken to perform inference.

Stan uses reverse-mode algorithmic differentiation to compute the gradient, which, according to Stan's Reference Manual, is done in two steps:

(1) Construct the expression graph that is used to define the target probability function.

(2) Evaluate the gradients by backpropagating partial derivatives along the expression graph.

The gradient computation time does not directly depend on the number of parameters of the model, but only on the size of the graph, or in other words — on the number of sub-expressions defining the target density. One way to reduce the gradient computation time is then to reduce the number of sub-expressions by *vectorising* statements where possible.

Suppose, for example, that we have a vector of N independent variables $\mathbf{x}$, s.t. $\forall n \in \{1, \ldots, N\}.x_n \sim \mathcal{N}(0, 1)$. This can be written as a **for** loop:

```
vector[N] x;
for (n in 1:N){
  x[n] ~ normal(0, 1);
}
```

However, using the more concise form of vectorised function application that Stan supports, will result in less sub-expressions, and therefore smaller graph. The code below results in an equivalent program, but faster gradient computation, and therefore, inference:

```
vector[N] x;
x ~ normal(0, 1);
```

Arguments can also be *lifted* to vectors, for example if we wanted the N variables $\mathbf{x}$ to have different means:

```
vector[N] x;
vector[N] mu;
x ~ normal(mu, 1);
```

Implementing the above as a **for** loop, stating `y[i] ~ normal(mu[i], 1)` for each `i`, will again result in an equivalent, but less efficient, Stan program.

Finally, arithmetic operations in Stan are also overloaded to work on vector and matrix objects. This means even more drastic reduction of sub-expressions in the target probability density. Below is an example taken from Stan's Reference Manual, illustrating two ways of writing an equivalent Linear Regression model, with N outcomes $\mathbf{y}$, and K predictors $\mathbf{x}_n$ for each outcome n. The model on the left defines the outcomes and predictors as arrays, and uses **for** loops to define the distribution each y_n comes from. The model on the right uses linear algebra objects and operations.

The code on the right-hand side is not only more concise, but it also results in a more efficient inference algorithm.

```
data {
  int K;
  int N;
  real X[K, N];
  real y[N];
}
parameters {
  real beta[K];
}
model {
  for (n in 1:N) {
    real gamma;
    gamma = 0;
    for (k in 1:K)
      gamma = gamma + X[n, k] *
          beta[k];
    y[n] ~ normal(gamma, 1);
  }
}
```

```
data {
  int K;
  int N;
  matrix[N, K] X;
  vector[N] y;
}
parameters {
  vector[K] beta;
}
model {
  y ~ normal(X * beta, 1);
}
```

Chapter 3

Aims, Contributions, and Approach

If we could imagine the perfect probabilistic programming language, it would be one where the programmer specifies a probabilistic model, without knowing anything about inference, yet that model is compiled to an efficient and optimised inference algorithm. In practise, this is very difficult. As a consequence, current probabilistic languages face a trade-off between usability and efficiency. Looking at one such popular and efficient language — Stan — we can identify several design choices made to make compilation to a Hamiltonian Monte Carlo sampling algorithm possible, at the expense of requiring more work or expertise from the programmer.

The goal of this project is to improve probabilistic programming by enhancing Stan by, for example, introducing new language constructs to facilitate different families of models, or using programming language techniques to reduce the work for the programmer. We identify the following weakness of Stan:

(1) **Lack of compositional syntax**

One major design choice, made in order to make compilation to an efficient Hamiltonian Monte Carlo algorithm possible, is the presence of *program blocks* (§§ 2.1.2). Data needs to be defined in one particular block, parameters in another; blocks must appear in order; and changing the block a variable is defined in could result in a semantically equivalent, but less or more efficient program. Stan allows a wide range of models to be defined in its modelling language, but requires knowledge, or at least insight, about the specifics of the underlying inference algorithm. Moreover, the presence of blocks makes it difficult to reason about a program as a composition of other Stan programs, which also may affect usability.

(2) **Inflexible user-defined functions**

The block structure of Stan's modelling language makes it difficult to implement flexible user-defined functions. In particular, user-defined functions in Stan cannot declare new parameters, making them inapplicable to a large set of desirable transformations (such as model reparameterisation). A recent talk by Gelman [2017] also calls for improving Stan in a direction allowing for more readable, understandable models that support better modulation and encapsulation.

(3) **No dedicated mixture models syntax**

Stan's underlying inference algorithm (Hamiltonian Monte Carlo) uses information about the gradient of the target density function, as previously discussed in § 2.2. This means that the probability density defined by Stan models needs to be *differentiable*, which makes the use of discrete parameters difficult. As a consequence, even though one can specify a *mixture model* in Stan, discrete parameters must be marginalised over by hand, by directly accessing the accumulated log probability density function (see §§ 2.1.1).

(4) **No dedicated hierarchical regression models syntax**

The original aim of Stan is to apply Bayesian inference to hierarchical (multilevel) linear models. Currently, a hierarchical model as simple as a multilevel linear regression requires the user to explicitly model different datapoints, using **for** loops and array indexing (see Appendix A). Although intuitive to the imperative programmer, this syntax leads to a lengthy and hard-to-read model specifications, as well as to code prone to errors.

3.1 Contributions

Towards point (1), we design the SlicStan language, which allows the interleaving of declarations and statements involving variables that would belong to different program blocks if the program was written in Stan. Excluding the program blocks, the syntax is similar to a subset of Stan. We make use of *information flow analysis* and *type inference* to deduce a *level type* of each variable (§§ 4.2.3). Using this information we are then able to translate a SlicStan program to a Stan program, by *shredding* variable declarations and statements into different program blocks.

Towards point (2), we extend SlicStan to support more flexible user-defined functions. Using the level type information obtained as described above, we are able to translate a SlicStan program to a Stan program in two steps:

(1) *Elaboration* (§§ 4.3.2) of the SlicStan program into a core SlicStan program that does not contain any blocks or function calls.

(2) *Transformation* (§§ 4.3.3) of the core SlicStan program into Stan.

To give an example of a SlicStan program, consider the model from §§ 2.1.2. We are able to translate the SlicStan program on the next page (*left*), to a well-formed, equivalent to that from Figure 2.1, Stan program (*right*):

SlicStan

```
data int N;
data real[N] y;
data real mu_mu;
data real sigma_mu;

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

real mu_y ~ normal(0, 1);

real sigma_y = pow(tau_y, -0.5);
y ~ normal(mu_y, sigma_y);

real variance_y = pow(sigma_y, 2);
```

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(0, 1);
  y ~ normal(mu_y, sigma_y);
}
generated quantities {
  real variance_y = pow(sigma_y,2);
}
```

An initial design towards point (3) has been drafted, and the underlying idea has been partially tested (§§ 4.5.3). We leave point (4) for future work.

The rest of this dissertation mainly focuses on our contribution towards point (1) — the automation of variable and statement assignments to blocks in Stan — and point (2) — the introduction of more flexible user-defined functions.

3.2 Approach

We look back at Stan's program blocks definitions (§§ 2.1.2) in attempt to formalise their usage. We define each block to have one of 3 *levels* as follows:

- The **data** and **transformed data** blocks are at level DATA.
- The **parameters**, **transformed parameters**, and **model** blocks are at level MODEL.

- The **generated quantities** block is at level GENQUANT

Knowing that blocks must appear in order in a Stan program, and that each block can only access variables defined in itself, or a previous block, we can see that *information flows* from DATA through MODEL to GENQUANT, but not in the reverse order. For example, a transformed parameter (level MODEL) can be defined in terms of a transformed data variable (level DATA), but cannot be defined in terms of a generated quantity (GENQUANT).

Now suppose we take a sequence of statements and variable declarations, where each variable is annotated with one of the 3 levels DATA, MODEL, or GENQUANT. We formalise the question of whether or not this sequence can be split into Stan blocks that obey the ordering from above, as an *information flow analysis* problem.

3.2.1 Secure Information Flow Analysis

Information flow is the transfer of information between two variables in some process. For example, if a program contains the statement `y = x + 1`, then information flows from `x` to `y`. Static analysis techniques concerning the flow of information are especially popular in the security community, where the goal is to prove that systems do not leak information.

Secure information flow analysis has a long history, summarised by Sabelfeld and Myers [2003], and Smith [2007]. It concerns systems where variables can be seen as having one of several *security levels*. For example, there could be two security levels:

- LOW — such variables hold *low* security, public data.
- HIGH — such variables hold *high* security, secret data.

We then want to disallow the flow of secret information to a public variable, but allow other flows of information. That is, if `L` is a variable of security level LOW, and `H` is a variable (or an expression) of security level HIGH, we want to forbid statements such as `L = H`, but allow:

- `L = L`
- `H = H`
- `H = L`

Formally, the levels form a *lattice*,[1] where

$$\text{LOW} \leq \text{HIGH}$$

Secure information flow analysis is then used to ensure that information flows only upwards that lattice. This is also known as the *noninterference property* — confidential data may not interfere with public data.

[1] A lattice is a partially ordered set $(L, \leq)$, where every two elements of L have a unique least upper bound and a unique greatest lower bound.

3.2.2 Information Flow Analysis of a SlicStan program

The key idea behind SlicStan is to use information flow analysis to find the "level" of variables in a probabilistic model, in order to translate the model to a Stan program that is optimised for Hamiltonian Monte Carlo inference.

We define the lattice $\{\{\text{DATA}, \text{MODEL}, \text{GENQUANT}\}, \leq\}$, where

$$\text{DATA} \leq \text{MODEL} \leq \text{GENQUANT}$$

We then implement SlicStan's type system (§ 4.2) according to Volpano et al. [1996], who show how to formalise the information flow problem as a type system, and prove that a well-typed program has the noninterference property. In other words, we implement the language and its type system, such that in a well-typed SlicStan program, information flows from variables of level DATA, through those of level MODEL, to those of level GENQUANT, but never in the opposite direction.

Going back to §§ 2.2.3, and Table 2.1, we identify another ordering on the level types, which orders them in terms of performance. Code associated with variables of level DATA is executed only once, code associated with variables of level GENQUANT — once per sample, and if the variables are of level MODEL — once per leapfrog (many times per sample). This means that there is the following ordering of level types in terms of performance:

$$\text{DATA} < \text{GENQUANT} < \text{MODEL}$$

We implement *type inference* (§§ 4.2.3), following the typing rules of SlicStan's type system, to obtain a level type for each variable in a SlicStan program, so that:

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

Having inferred a level type for each program variable, we can use the following rules to determine what block of a Stan program a declaration of a variable x, and statements associated with it, should belong to:

- x is of level DATA
  - x has *not* been assigned to — x belongs to the **data** block.
  - x has been assigned to — x belongs to the **transformed data** block.
- x is of level MODEL
  - x has *not* been assigned to — x belongs to the **parameters** block.
  - x has been assigned to — x belongs to the **transformed parameters** block.
- x is of level GENQUANT
  - x belongs to the **generated quantities** block.

The full set of formal rules that specify how a SlicStan program is translated to a Stan program, is given in §§ 4.3.3.

3.2.3 Extending SlicStan with User-defined functions

As discussed previously, due to Stan's block system, the functionality of its user-defined functions is constrained. More specifically, it is possible to define local variables, and it is also possible to increment the log probability **target** variable in Stan, for example:

```
void normal_5_lp(real x){
  real m = 5;
  x ~ normal(m, 1);
}
```

is a legal Stan function. However, *reparameterisation* by using a function is not possible if it involves the declaration of a new parameter. For example:

```
real normal_m_lp(real m){
  real x;
  x ~ normal(0, 1);
  return (x + m);
}
```

will result in a runtime initialisation error if called, as x is not a parameter, meaning that x must be initialised, which is not. Even if we initialise x after declaring it in the body of normal_m, we can only call the function from the **model** block, and thus we can only use its return value locally within the same block. A reparameterisation such as the one shown in the *Neal's Funnel* example (§§ 2.3.2) cannot be expressed in terms of the user-defined functions that Stan currently supports.

As SlicStan is not constrained to have its variable declarations in particular blocks, it is also able to support more flexible user-defined functions. We extend SlicStan's syntax, typing rules, and translation rules accordingly, and demonstrate that SlicStan programs containing flexible[2] user-defined functions translate to well-formed Stan programs. Reparameterising Neal's Funnel using functions is then possible, with the code below being a well-typed SlicStan program expressing it:

```
def my_normal(real m, real s){
  real xr;
  xr ~ normal(0, 1);
  return s * xr + m;
}
real y = my_normal(0, 3);
real z = my_normal(0, exp(y * 0.5));
```

[2]Compared to functions in the current version of Stan. There are still limitations, for example recursive functions are not supported.

Chapter 4

Design and Implementation of SlicStan

This chapter describes in detail the design of SlicStan — a probabilistic programming language inspired by Stan, which attempts to improve on Stan's usability, while still producing optimised inference code. We use static analysis techniques to reason about what class (e.g. data, parameter) a variable belongs to, and thus show that Stan's block syntax is not a necessary element needed for the program to be compiled for Hamiltonian Monte Carlo inference. Moreover, eliminating the need for the programmer to structure their code in program blocks allows us to introduce more flexible user-defined functions.

In short (with Chapter 3 giving a longer overview) SlicStan aims to improve probabilistic programming by using existing programming language techniques, and in particular — *information flow analysis*. This requires formalising the language's syntax (§ 4.1), and specifying *typing rules* (§ 4.2), which can be used to infer the *level types* of variables. SlicStan can then be translated to optimised Stan code (§ 4.3). A parser, type-checker and compiler for SlicStan were implemented in F# (§ 4.4), with Chapter 5 demonstrating the capabilities of this language implementation. We also discuss and provide a preliminary design of a few additional features (§ 4.5).

4.1 Syntax of SlicStan

SlicStan's syntax follows closely that of (a subset of) Stan, excluding the presence of program blocks. In this section, we give a formal syntax, and informal semantics of the language.

A SlicStan program is a sequence of function definitions F_n, and a single statement S:

Syntax of a SlicStan Program

$P ::= F_1, \ldots, F_N, S \quad N \geq 0$	SlicStan program

The building blocks of a SlicStan statement are expressions, l-values, and distributions.

Expressions cover most of Stan's expressions, including variables, constants, arrays and array elements, and function calls. One difference with Stan is that in SlicStan a difference is made between a call to a *primitive function* (f), and a call to a *user-defined function* (F_e), as those are treated differently during the type-inference and translation stages. Similarly, there is a difference between a *primitive distribution* (d), and a *user-defined distribution* (F_d). L-values denote actual *locations* in memory.

Syntax of Expressions, Distributions, and L-Values:

$E ::=$	expression
x	variable
c	constant
$[E_1, ..., E_N]$	array
$E_1[E_2]$	array element
$f(E_1, \ldots, E_N)$	primitive expression function call[1]
$F_e(E_1, \ldots, E_N)$	user-defined expression function call
$D ::=$	parametric distribution
$d(E_1, \ldots, E_N)$	primitive distribution function call
$F_d(E_1, \ldots, E_N)$	user-defined distribution function call
$L ::=$	L-value
x	variable
$L[E]$	array element

SlicStan supports only a subset of Stan's statements. For example, SlicStan does not currently support **for** or **while** loops, nor **if** statements. The assignment ($L = E$), model ($E \sim D$), and sequence ($S_1; S_2$) statements have the same meaning as in Stan, with only l-values (L) appearing on the left-hand side of an assignment.

Syntax of Statements

$S ::=$	statement
data $\tau\, x; S$	data declaration
$L = E$	assignment
$E \sim D$	model
$S_1; S_2$	sequence
$F_v(E_1, \ldots, E_N)$	void function call
$\{Tx; S\}$	block statement
skip	skip

There are two statements that are different to those of Stan. Firstly, the *data declaration statement* declares a new variable, stating explicitly that the value of this variable won't be known at compile time, but it will be received as an input in order to run the program. This statement is essentially equivalent to stating that the variable it declares is in Stan's

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

data block. This declaration is needed, in order to distinguish data from parameters in the probabilistic program, as both of those are unknown at compile time. Secondly, the *block statement* defines a new variable, together with its type and scope. The *base types* τ, and *full types* T take values as described in the next section.

Finally, a function definition consists of an identifier f, a list of arguments and their types $T_i\, a_i$, a statements S, and a return value E, D, or void.

Syntax of Function Definitions

$F ::=$	function definition
def $f(T_1\, a_1, \ldots, T_N\, a_N)\ S$ **return** E	expression return type definition
def $f(T_1\, a_1, \ldots, T_N\, a_N)\ S$ **return** D	distribution return type definition
def $f(T_1\, a_1, \ldots, T_N\, a_N)\ S$	void return type definition

Both build-in and user-defined functions are associated with function signatures $\langle \vec{\tau} \to \tau \rangle$, where $\vec{\tau}$ abbreviates $\tau_1, \ldots, \tau_N$.

4.2 Typing of SlicStan

This section introduces the type system of SlicStan. It presents the declarative (§§ 4.2.1), and algorithmic (§§ 4.2.2) typing rules of the language, and it shows how the latter are used to implement type inference (§§ 4.2.3).

Types T in SlicStan range over pairs (τ, ℓ) of a base type τ (which can be any of Stan's (unconstrained) types), and a level type ℓ — one of DATA, MODEL, or GENQUANT. Arrays, vectors and matrices are sized, with $n, m \geq 0$.

Types, and Type Environment:

$\ell ::=$	level type
DATA	data, transformed data
MODEL	parameters, transformed parameters
GENQUANT	generated quantities
$n, m ::=$	size
$\tau ::=$	base type
real | **int** | **vector**$[n]$ | **matrix**$[n, m]$ |	basic types
$\tau[n]$	array
$T ::= (\tau, \ell)$	type: base type and level
$\Gamma ::= x_1 : T_1, \ldots, x_N : T_N \quad x_n$ distinct	typing environment

4.2.1 Declarative Typing Rules

Each judgment of the type system is with respect to a typing environment Γ, which is defined to be a mapping from variable names to their full types, as above. We define

the following typing judgments for expressions, l-values, distributions, and statements respectively:

Judgments of the Type System:

$\Gamma \vdash E : (\tau, \ell)$	expression E has type (τ, ℓ)
$\Gamma \vdash L : (\tau, \ell)$	l-value L has type (τ, ℓ)
$\Gamma \vdash D : (\tau, \ell)$	distribution of a variable of type (τ, ℓ)
$\Gamma \vdash S : \ell$	statement S assigns only to level ℓ and above

A top-level SlicStan program consists of a typing environment Γ and a statement S such that $\Gamma \vdash S : \text{DATA}$. This means, that S can contain assignments to variables at level DATA and above.

We now present the full set of declarative typing rules, which follow closely 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 using the subsumption rules (ESub), (DSub), and (SSub), which together allow information to only flow upwards the DATA $\leq$ MODEL $\leq$ GENQUANT lattice.

The rules for expressions and l-values follow a standard pattern, with two exceptions. One is the presence of the *vectorised* relation $\mathcal{V}$ in the (EPrim) and (ECall) rules. The relation is defined as stated in Definition 1, and aims to legalise *vectorised calls* to functions in SlicStan, in a similar way to what is currently supported in Stan. Secondly, in (ECall), and later (DCall) and (VCall), we use an extended signature for the user-defined function $F_e : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\tau, \ell)$, which includes level types. This is in order to allow for more flexible type-inference later.

Typing Rules for Expressions:

(ESub)
$$\frac{\Gamma \vdash E : (\tau, \ell) \quad \ell \leq \ell'}{\Gamma \vdash E : (\tau, \ell')}$$

(Var)
$$\frac{}{\Gamma, x : T \vdash x : T}$$

(Const)
$$\frac{\mathbf{ty}(c) = \tau}{\Gamma \vdash c : (\tau, \text{DATA})}$$

(Arr)
$$\frac{\Gamma \vdash E_n : (\tau, \ell) \quad \forall n \in 1..N}{\Gamma \vdash [E_1, ..., E_N] : (\tau[\,], \ell)}$$

(ArrEl)
$$\frac{\Gamma \vdash E_1 : (\tau[\,], \ell) \quad \Gamma \vdash E_2 : (\mathbf{int}, \ell)}{\Gamma \vdash E_1[E_2] : (\tau, \ell)}$$

(EPrim)$(f : \tau_1, \ldots, \tau_N \to \tau)$
$$\frac{\Gamma \vdash E_n : (\tau'_n, \ell) \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau'} \to \tau')}{\Gamma \vdash f(E_1, \ldots, E_N) : (\tau', \ell)}$$

(ECall)$(F_e : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\tau, \ell))$
$$\frac{\Gamma \vdash E_n : (\tau'_n, \ell_n) \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau'} \to \tau')}{\Gamma \vdash F_e(E_1, \ldots, E_N) : (\tau', \ell)}$$

Typing Rules for L-Values:

(LVar)
$$\frac{}{\Gamma, x : T \vdash x : T}$$

(LArrEl)
$$\frac{\Gamma \vdash L : (\tau[\,], \ell) \quad \Gamma \vdash E : (\mathbf{int}, \ell)}{\Gamma \vdash L[E] : (\tau, \ell)}$$

Definition 1. *Let $\mathcal{V}(\langle \vec{\tau} \to \tau \rangle, \langle \vec{\tau}' \to \tau' \rangle)$ be a predicate on pairs of type signatures, such that it is true if and only if the type signature $\langle \vec{\tau} \to \tau \rangle$ can be lifted to the type signature $\langle \vec{\tau}' \to \tau' \rangle$. Formally, $\mathcal{V}(\langle \vec{\tau} \to \tau \rangle, \langle \vec{\tau}' \to \tau' \rangle)$ is true if and only if:*

$$[\, \vec{\tau} = \vec{\tau}' \land \tau = \tau' \,] \lor [\, \exists k.\, Q(k, \vec{\tau}, \vec{\tau}') \land L(k, \tau, \tau') \,]$$

Where $Q(k, \vec{\tau}, \vec{\tau}') = Q(k, \tau_1, \ldots, \tau_N, \tau'_1, \ldots, \tau'_N)$ is a predicate which is true if and only if for all pairs of types τ_n and τ'_n, it's either the case that τ'_n is the type τ_n lifted, or the two types are equal. Formally:

$$Q(k, \vec{\tau}, \vec{\tau}') = \forall n. L(k, \tau_n, \tau'_n) \lor \tau_n = \tau'_n$$

And $L(k, \tau, \tau')$ is a predicate defined on types, which is true whenever τ' is a lifted version of τ of size k. Formally:

$$\begin{aligned} L(k, \tau, \tau') = {} & (\, \tau' = \tau[k] \,) \lor \\ & (\, \tau = \textbf{real} \land \tau' = \textbf{vector}[k]) \lor \\ & \exists m. \, (\, \tau = \textbf{vector}[m] \land \tau' = \textbf{matrix}[m, k]) \end{aligned}$$

Similarly to the rules for expressions, the rules for densities include vectorisation through $\mathcal{V}$ and extended signatures for user-defined densities $F_d : (\tau_1, \ell_1), \ldots, (\tau_n, \ell_n) \to (\tau, \ell)$.

Typing Rule for Distributions:

$$\frac{\Gamma \vdash D : (\tau, \ell) \quad \ell \leq \ell'}{\Gamma \vdash D : (\tau, \ell')} \text{(DSub)} \qquad \frac{\Gamma \vdash E_i : (\tau'_i, \ell) \quad \forall i \in 1..n \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau')}{\Gamma \vdash d(E_1, \ldots, E_n) : (\tau', \ell)} \text{(DPrim)}(d : \tau_1, \ldots, \tau_n \to \tau)$$

$$\frac{\Gamma \vdash E_i : (\tau'_i, \ell_i) \quad \forall i \in 1..n \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau')}{\Gamma \vdash F_d(E_1, \ldots, E_n) : (\tau', \ell)} \text{(DCall)}(F_d : (\tau_1, \ell_1), \ldots, (\tau_n, \ell_n) \to (\tau, \ell))$$

The rules for statements have two focal points: the (Model) and (Block) rules. As described previously, the model statement $E \sim D$ can be seen as an increment to the **target** variable that holds the accumulated log probability density function. This variable is in its nature of level MODEL — it can only be accessed in Stan's **model** block. The assignment to this variable must be a statement of level MODEL too, and thus we can only typecheck $\Gamma \vdash E \sim D :$ MODEL. Generated quantities (level GENQUANT) cannot, by definition, affect the target density, meaning that both E and D must be at most at MODEL level.

The (Block) rule concludes a judgment involving the definition of a variable x with scope S through a block statement. We differentiate between the cases where x has been assigned to in S, and the cases when it has not. This is determined by the *assigned* set $A(S)$ (Definition 2). Unassigned variables, in both Stan and SlicStan, are either unknown

(at compile time) data, parameters of the model, or unused generated quantities. However, data must be specified using the *data declaration* statement, $(\textbf{data}\,\tau\,x; S)$, meaning that unassigned variables declared using a block statement must be of at least level MODEL. That is either $x \in A(S)$ or $\text{MODEL} \leq \ell$, as stated by (BLOCK).

Typing Rules for Statements:

$$\frac{\Gamma \vdash S : \ell' \quad \ell \leq \ell'}{\Gamma \vdash S : \ell}\ (\text{SSUB}) \qquad \frac{\Gamma \vdash L : (\tau, \ell) \quad \Gamma \vdash E : (\tau, \ell)}{\Gamma \vdash (L = E) : \ell}\ (\text{ASSIGN}) \qquad \frac{\Gamma \vdash E : (\tau, \text{MODEL}) \quad \Gamma \vdash D : (\tau, \text{MODEL})}{\Gamma \vdash (E \sim D) : \text{MODEL}}\ (\text{MODEL})$$

$$\frac{\Gamma, x : (\tau, \ell) \vdash S : \ell' \quad (\text{MODEL} \leq \ell) \vee x \in A(S)}{\Gamma \vdash \{(\tau, \ell)\, x; S\} : \ell'}\ (\text{BLOCK}) \qquad \frac{\Gamma, x : (\tau, \text{DATA}) \vdash S : \ell}{\Gamma \vdash (\textbf{data}\ \tau\ x; S) : \ell}\ (\text{DATA})$$

$$\frac{\Gamma \vdash S_1 : \ell \quad \Gamma \vdash S_2 : \ell}{\Gamma \vdash (S_1; S_2) : \ell}\ (\text{SEQ}) \qquad \frac{}{\Gamma \vdash \mathit{skip} : \ell}\ (\text{SKIP}) \qquad \frac{\Gamma \vdash E_n : (\tau'_n, \ell_n) \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \mathsf{void}, \vec{\tau'} \to \mathsf{void})}{\Gamma \vdash F_v(E_1, \ldots, E_N) : \ell}\ (\text{VCALL})(F_v : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\mathsf{void}, \ell))$$

Definition 2 (Assigned set $A(S)$)**.** *$A(S)$ is the set that contains the names of variables that have been assigned to within the statement S. It is defined recursively as follows:*

$$A(x = E) = \{x\} \qquad A(S_1; S_2) = A(S_1) \cup A(S_2)$$
$$A(E \sim D) = \varnothing \qquad A(\mathit{skip}) = \varnothing$$
$$A(\{Tx; S\}) = A(S) \setminus \{x\} \qquad A(F_v(E_1, \ldots, E_n)) = \varnothing$$
$$A(\textbf{data}\ \tau\ x; S) = A(S)$$

4.2.2 Algorithmic Typing Rules

In this section, we present the algorithmic typing rules that SlicStan's *bidirectional* type checker implements.

Bidirectional type checking [Pierce and Turner, 2000] is a technique for developing a simple and efficient type checking algorithm, that produces informative type error messages. The key idea behind the method is to define *two* separate typing relations — one for type checking ($\leftarrow$), and one for type synthesis ($\rightarrow$). Throughout the algorithm, we use the former when we expect an expression or statement to have a certain type, while we use the latter to try and synthesise that type in case we do not know it.

In the case of SlicStan, this gives rise to the following judgments of the type system:

Judgments of the Algorithmic Type System:

$\Gamma \vdash E \rightarrow (\tau, \ell)$	expression E synthesizes type (τ, ℓ)
$\Gamma \vdash E \leftarrow (\tau, \ell)$	expression E checks against type (τ, ℓ)
$\Gamma \vdash L \rightarrow (\tau, \ell)$	l-value L synthesizes type (τ, ℓ)
$\Gamma \vdash D \rightarrow (\tau, \ell)$	distribution D synthesizes type (τ, ℓ)
$\Gamma \vdash D \leftarrow (\tau, \ell)$	distribution D checks against type (τ, ℓ)
$\Gamma \vdash S \rightarrow \ell, \Gamma'$	statement S synthesizes type level ℓ, and defines the set of variables Γ'
$\Gamma \vdash S \leftarrow \ell$	statement S checks against type level ℓ

To specify the algorithmic rules, we use the fact that the 3 level types, DATA, MODEL, GENQUANT, and their partial order $\leq$, form a lattice (§ 3.2), and define:

- $\ell_1 \sqcup \ell_2$ to be the unique least upper bound of ℓ_1 and ℓ_2,
- $\ell_1 \sqcap \ell_2$ to be the unique greatest lower bound of ℓ_1 and ℓ_2,
- $\bigsqcup_{i=1}^{I} \ell_i = (((\ell_1 \sqcup \ell_2) \sqcup \ell_3) \sqcup ...) \sqcup \ell_I$, and
- $\bigsqcap_{i=1}^{I} \ell_i = (((\ell_1 \sqcap \ell_2) \sqcap \ell_3) \sqcap ...) \sqcap \ell_I$.

The algorithmic rules then follow closely the declarative ones, with (ESwap), (DSwap), and (SSwap) testing for subsumption. The interest here is mainly on the rules for calls to user-defined functions: (Synth ECall), (Synth DCall), and (Synth VCall). Each of them ensures the subtyping is:

- *Contravariant* in the arguments of the function. This is done by including the premise $\ell'_i \leq \ell_i$ in each rule, where ℓ_i is the level of the i^{th} argument of the function, and ℓ'_i is the level of the i^{th} expression passed as an argument to the function.
- *Covariant* in the return type of the function, which is done by the presence of the swap rules.

Algorithmic Typing Rules for Expressions:

$$\text{(ESwap)}\quad \frac{\Gamma \vdash E \to (\tau, \ell') \quad \ell' \leq \ell}{\Gamma \vdash E \leftarrow (\tau, \ell)} \qquad \text{(Synth Var)}\quad \frac{\Gamma(x) = T}{\Gamma \vdash x \to T} \qquad \text{(Synth Const)}\quad \frac{\mathbf{ty}(c) = \tau \quad \ell = \text{DATA}}{\Gamma \vdash c \to (\tau, \ell)}$$

$$\text{(Synth Arr)}\quad \frac{\Gamma \vdash E_n \to (\tau_n, \ell_n) \quad \tau_n = \tau \quad \forall n \in 1..N}{\Gamma \vdash [E_1, ..., E_N] \to (\tau[\,], \bigsqcup_n \ell_n)}$$

$$\text{(Synth ArrEl)}\quad \frac{\Gamma \vdash E_1 \to (\tau_1, \ell_1) \quad \tau_1 = \tau[\,] \quad \Gamma \vdash E_2 \to (\tau_2, \ell_2) \quad \tau_2 = \mathbf{int}}{\Gamma \vdash E_1[E_2] \to (\tau, \ell_1 \sqcup \ell_2)}$$

$$\text{(Synth EPrim)}(f : \tau_1, \ldots, \tau_N \to \tau)\quad \frac{\Gamma \vdash E_n \to (\tau'_n, \ell_n) \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau')}{\Gamma \vdash f(E_1, \ldots, E_N) \to (\tau', \bigsqcup_n \ell_n)}$$

$$\text{(Synth ECall)}(F_e : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\tau, \ell))\quad \frac{\Gamma \vdash E_i \to (\tau'_n, \ell'_n) \quad \ell'_n \leq \ell_n \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau')}{\Gamma \vdash F_e(E_1, \ldots, E_N) \to (\tau', \ell)}$$

Algorithmic Typing Rules for L-Values:

$$\text{(Synth LVar)}\quad \frac{\Gamma(x) = T}{\Gamma \vdash x \to T} \qquad \text{(Synth LArrEl)}\quad \frac{\Gamma \vdash L \to (\tau_L, \ell_L) \quad \tau_L = \tau[\,] \quad \Gamma \vdash E \to (\tau_E, \ell_E) \quad \tau_E = \mathbf{int}}{\Gamma \vdash L[E] \to (\tau, \ell_L \sqcup \ell_E)}$$

Algorithmic Typing Rule for Distributions:

$$\text{(DSwap)}\quad \frac{\Gamma \vdash D \to (\tau, \ell') \quad \ell' \leq \ell}{\Gamma \vdash D \leftarrow (\tau, \ell)} \qquad \text{(Synth DPrim)}(d : \tau_1, \ldots, \tau_N \to \tau)\quad \frac{\Gamma \vdash E_n \to (\tau'_n, \ell_n) \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau') \quad \forall n \in 1..N}{\Gamma \vdash d(E_1, \ldots, E_N) \to (\tau', \bigsqcup_n \ell_n)}$$

$$\text{(Synth DCall)}(F_d : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\tau, \ell))\quad \frac{\Gamma \vdash E_n \to (\tau'_n, \ell'_n) \quad \ell'_n \leq \ell_n \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau}' \to \tau')}{\Gamma \vdash F_d(E_1, \ldots, E_N) \to (\tau', \ell)}$$

The algorithmic typing rules for statements S contain one more detail that is not present in the declarative ones — the "return" environment Γ, which contains the variables declared in S, together with their types. This detail is needed in order to correctly synthesise user-defined functions' signatures, as shown below.

Algorithmic Typing Rules for Statements:

(SSwap)
$$\frac{\Gamma \vdash S \to \ell', \Gamma' \quad \ell \leq \ell'}{\Gamma \vdash S \leftarrow \ell}$$

(Synth Assign)
$$\frac{\Gamma \vdash L \to (\tau, \ell) \quad \Gamma \vdash E \leftarrow (\tau, \ell)}{\Gamma \vdash (L = E) \to \ell, \varnothing}$$

(Synth Model)
$$\frac{\Gamma \vdash E \to (\tau, \ell) \quad \ell \leq \textsc{model} \quad \Gamma \vdash D \leftarrow (\tau, \textsc{model})}{\Gamma \vdash (E \sim D) \to \textsc{model}, \varnothing}$$

(Synth Block)
$$\frac{\Gamma, x : T \vdash S \to \ell, \Gamma' \quad (\textsc{model} \leq \ell) \vee x \in A(S)}{\Gamma \vdash \{Tx; S\} \to \ell, (\Gamma', x : T)}$$

(Synth Data)
$$\frac{\Gamma, x : (\tau, \textsc{data}) \vdash S \to \ell, \Gamma'}{\Gamma \vdash (\textbf{data}\ \tau\ x; S) \to \ell, (\Gamma', x : (\tau, \textsc{data}))}$$

(Synth Seq)
$$\frac{\Gamma \vdash S_1 \to \ell_1, \Gamma_1 \quad \Gamma \vdash S_2 \to \ell_2, \Gamma_2}{\Gamma \vdash (S_1; S_2) \to \ell_1 \sqcap \ell_2, (\Gamma_1, \Gamma_2)}$$

(Synth Skip)
$$\frac{\ell = \textsc{genquant}}{\Gamma \vdash \mathit{skip} \to \ell, \varnothing}$$

(Synth VCall)$(F_v : (\tau_1, \ell_1), \ldots, (\tau_N, \ell_N) \to (\mathsf{void}, \ell))$
$$\frac{\Gamma \vdash E_n \to (\tau'_n, \ell'_n) \quad \ell'_n \leq \ell_n \quad \forall n \in 1..N \quad \mathcal{V}(\vec{\tau} \to \tau, \vec{\tau'} \to \tau')}{\Gamma \vdash F_v(E_1, \ldots, E_N) \to \ell, \varnothing}$$

Finally, a SlicStan program, which consists of a list of function definitions $F_1, \ldots, F_N$, and a main body S, typechecks if:

- All user-defined functions F_i typecheck, and
- S assigns to variables of level DATA and above.

Algorithmic Typing Rule for Function Definitions:

(Synth VDef)
$$\frac{a_1 : T_1, \ldots, a_N : T_N \vdash S \to \ell, \Gamma}{\varnothing \vdash \textbf{def}\ F_V(T_1\, a_1, \ldots, T_N\, a_N)\ S \to (T_1, \ldots, T_N \to (\mathsf{void}, \ell))}$$

(Synth EDef)
$$\frac{a_1 : T_1, \ldots, a_N : T_N \vdash S \to \ell, \Gamma \quad a_1 : T_1, \ldots, a_N : T_N, \Gamma \vdash E \to \tau, \ell'}{\varnothing \vdash \textbf{def}\ F_E(T_1\, a_1, \ldots, T_N\, a_N)\ S\ \textbf{return}\ E \to (T_1, \ldots, T_N \to (\tau, \ell'))}$$

(Synth DDef)
$$\frac{a_1 : T_1, \ldots, a_N : T_N \vdash S \to \ell, \Gamma \quad a_1 : T_1, \ldots, a_N : T_N, \Gamma \vdash D \to \tau, \ell'}{\varnothing \vdash \textbf{def}\ F_D(T_1\, a_1, \ldots, T_N\, a_N)\ S\ \textbf{return}\ D \to (T_1, \ldots, T_N \to (\tau, \ell'))}$$

Algorithmic Typing Rule for a SlicStan Program:

(CHECK SLICSTAN)

$$\frac{F_i \to \sigma_i \quad \forall i \in 1..N \quad \varnothing \vdash S \leftarrow \text{DATA}}{\varnothing \vdash \mathit{SlicStan}(F_1, \ldots, F_N, S) \leftarrow \text{DATA}}$$

Note that in all typing rules so far, we consider the signatures σ, of both primitive and user-defined functions, available. This is done for simplicity of the rules, as the set of signatures stays constant during type checking / translating the program (there are no local functions; all functions are declared as a list in the beginning of the program).

Properties of the Algorithmic Type System

Having defined the algorithmic type system, we are interested in proving its soundness with respect to the declarative typing system from §§ 4.2.1.

Conjecture 1 (Soundness of the Algorithmic Type System)**.** *For all statements S, typing environments Γ and Γ', expressions E, l-values L, distributions D, base types τ, and level types ℓ:*

(1) *If* $\Gamma \vdash E \to (\tau, \ell)$, *then* $\Gamma \vdash E : (\tau, \ell)$.

(2) *If* $\Gamma \vdash E \leftarrow (\tau, \ell)$, *then* $\Gamma \vdash E : (\tau, \ell)$.

(3) *If* $\Gamma \vdash L \to (\tau, \ell)$, *then* $\Gamma \vdash L : (\tau, \ell)$.

(4) *If* $\Gamma \vdash D \to (\tau, \ell)$, *then* $\Gamma \vdash D : (\tau, \ell)$.

(5) *If* $\Gamma \vdash D \leftarrow (\tau, \ell)$, *then* $\Gamma \vdash D : (\tau, \ell)$.

(6) *If* $\Gamma \vdash S \to \ell, \Gamma'$, *then* $\Gamma \vdash S : \ell$.

(7) *If* $\Gamma \vdash S \leftarrow \ell$, *then* $\Gamma \vdash S : \ell$.

We leave proving Conjecture 1 by structural induction for future work.

4.2.3 Level Type Inference

SlicStan's compiler fully infers the level types of variables, so that no information regarding what Stan block each of them belongs to needs to be provided by the programmer (other than what variables will be input as data, using the **data** $\tau\, x; S$ statement). This is done in three steps:

(1) **Introducing Level Type Placeholders**

Variables declared through the data statement **data** $\tau\, x; S$ are of level DATA. All other variables are declared using the block statement $\{(\tau, \ell_x)\, x; S\}$, where ℓ_x is a

placeholder for the actual level type of x. In SlicStan's implementations, placeholders are introduced by the language's parser, when constructing the abstract syntax tree of the program (see § 4.4). Note that the base type of x needs to be specified — SlicStan does not at the moment implement type inference for the base types τ.

(2) **Constraint Generation**

Constraint generation is done by applying the algorithmic typing rules to a program with level type placeholders, and emitting an inequality constraint every time a check needs to be done on unknown level types. For example, suppose that we want to infer the level types of x and y in the following statement:

$$S = \{(\textbf{real}, \ell_x)\, x, \{(\textbf{real}, \ell_y)\, y, x = 5; y = x\}\}$$

Starting from the conclusion $\varnothing \vdash S \leftarrow \textsc{data}$, we apply the rules (SSwap), 2× (Synth Block), (Synth Seq), to obtain the constraints:

- $\textsc{data} \leq \ell_x$ from applying (Synth Assign) and (ESwap) to

$$x : (\textbf{real}, \ell_x), y : (\textbf{real}, \ell_y) \vdash x = 5 \rightarrow \ell_x$$

- $\ell_x \leq \ell_y$ from applying (Synth Assign) and (ESwap) to

$$x : (\textbf{real}, \ell_x), y : (\textbf{real}, \ell_y) \vdash y = x \rightarrow \ell_y$$

(3) **Resolving the Constraints**

In most programs, there exist several possible solutions that satisfy the generated constraints. We would like to resolve them with respect to the program efficiency ordering described previously — taking variables to be at level DATA where possible, results in a faster Stan code than taking them to be at level GENQUANT, which in turn is faster than taking them to be at level MODEL:

$$\textsc{data} < \textsc{genquant} < \textsc{model}$$

We seek the most efficient assignment of level types to variables in three phases:

(a) Find the highest possible level for each variable, with respect to the given constraints and the partial order on levels $\leq$.

(b) Find the lowest possible level for each variable, with respect to the given constraints and the partial order on levels $\leq$.

(c) For each variable, choose the lowest of the two levels, with respect to the efficiency order $<$.

Looking back a the $S = \{(\textbf{real}, \ell_x)\, x, \{(\textbf{real}, \ell_y)\, y, x = 5; y = x\}\}$ example, we firstly find $\ell_x = \textsc{genquant}, \ell_y = \textsc{genquant}$, then $\ell_x = \textsc{data}, \ell_y = \textsc{data}$, and finally we chose the "lower" with respect to $<$ of the two assignments: $\ell_x = \textsc{data}, \ell_y = \textsc{data}$.

4.3 Translation of SlicStan to Stan

To implement translation to Stan, we firstly formally define the syntax of a subset of the Stan language, and state what a well-formed Stan program is (§§ 4.3.1). Translating a SlicStan program into a Stan program is then done in two steps:

(1) *Elaboration* of the SlicStan program into a *core* SlicStan program that does not contain any blocks or calls to user-defined functions (§§ 4.3.2).

(2) *Translation* of the elaborated program into Stan (§§ 4.3.3).

4.3.1 Formalisation of Stan

We define Stan's expressions, l-values, distributions and statements to be equivalent to those of a *core* SlicStan language, that does not have data statements, blocks, or user-defined-function calls. In other words, the building blocks of the subset of Stan we will work with are as below:

Syntax of Stan's Language Constructs:

$E ::=$	expression
$\quad x$	$\quad$ variable
$\quad c$	$\quad$ constant
$\quad [E_1, ..., E_n]$	$\quad$ array
$\quad E_1[E_2]$	$\quad$ array element
$\quad f(E_1, \ldots, E_n)$	$\quad$ expression function call
$D ::=$	distribution
$\quad d(E_1, \ldots, E_n)$	$\quad$ distribution function call
$L ::=$	l-values
$\quad x$	$\quad$ variable
$\quad L[E]$	$\quad$ array element
$S ::=$	statement
$\quad L = E$	$\quad$ assignment
$\quad E \sim D$	$\quad$ model
$\quad S_1; S_2$	$\quad$ sequence

Stan's (unconstrained) types τ are exactly equivalent to the base types in SlicStan. The typing environment Γ is, as before, a mapping between variable names and (Stan) types:

Types in Stan

$n, m ::=$	size
$\tau ::=$	type
$\quad$ **real** | **int** | **vector**$[n]$ | **matrix**$[n, m]$ | $\tau[n]$	

Apart from expressions, l-values, distributions, and statements, Stan's programs also contain declarations of variables. In the subset of Stan we choose (no declarations of local variables are possible), variable declarations must appear before statements in the program blocks. Therefore, we can say that a Stan program consists of 6 blocks, each with a typing environment or a typing environment and a statement associated with it:

data { Γ_d }
transformed data { Γ'_d S'_d }
parameters { Γ_m }
transformed parameters { Γ'_m S'_m }
model { Γ''_m S''_m }
generated quantities { Γ_q S_q }

In Stan, this typing environment will be a finite mapping between variable names and Stan types τ, however we extend this definition to allow for more convenient notation, by extending it to be a SlicStan typing environment. We also define the relation $\Gamma \vdash \ell$, to mean that all variables in Γ are of level ℓ.

Definition 3. *If* $\Gamma = x_1 : (\tau_1, \ell_1), \ldots, x_N : (\tau_N, \ell_N)$ *is a typing environment,* $\Gamma \vdash \ell$ *if and only if* $\forall i \in 1..N.(\ell_i = \ell)$.

Finally, given a Stan program P of the above form, we define it to be *well-formed*, written $\vdash_{Stan} P$, in terms of the declarative typing relation $\Gamma \vdash S : \ell$, as follows:

(1) $\Gamma_d \vdash$ DATA

(2) $\Gamma'_d \vdash$ DATA and $\Gamma_d, \Gamma'_d \vdash S'_d :$ DATA

(3) $\Gamma_m \vdash$ MODEL

(4) $\Gamma'_m \vdash$ MODEL and $\Gamma_d, \Gamma'_d, \Gamma_m, \Gamma'_m \vdash S'_m :$ MODEL

(5) $\Gamma''_m \vdash$ MODEL and $\Gamma_d, \Gamma'_d, \Gamma_m, \Gamma'_m, \Gamma''_m \vdash S''_m :$ MODEL

(6) $\Gamma_q \vdash$ GENQUANT and $\Gamma_d, \Gamma'_d, \Gamma_m, \Gamma'_m, \Gamma_q \vdash S_q :$ GENQUANT

The rest of this section explains how a (well-typed) SlicStan program is translated to a well-formed Stan program, in two steps: *elaboration* and *transformation*.

4.3.2 Elaboration

A SlicStan program $F_1, \ldots, F_N, S$ is *elaborated* to a pair $\langle \Gamma,\ S' \rangle$ of a *context* Γ and a SlicStan statement S', where S' does not contain any data declarations, blocks or function calls. In other words, S' is also a legal Stan statement. The context Γ is a typing environment containing all variables that are bound in S', together with their type.

The elaboration relation $\Downarrow$ is defined on pairs between an expression, l-value, distribution, statement, or a program, and a pair $\langle \Gamma,\ S \rangle$, $\langle \Gamma,\ S.E \rangle$, $\langle \Gamma,\ S.L \rangle$, or $\langle \Gamma,\ S.D \rangle$. The construct after the dot plays the role of a return type, and is used, together with the preceding statement S, to statically unroll all calls to user-defined functions.

Elaboration Relation

$E \Downarrow \langle \Gamma,\ S.E' \rangle$	expression elaboration
$L \Downarrow \langle \Gamma,\ S.L' \rangle$	l-value elaboration
$D \Downarrow \langle \Gamma,\ S.D' \rangle$	distribution elaboration
$S \Downarrow \langle \Gamma,\ S' \rangle$	statement elaboration
$F \Downarrow \langle A, \Gamma,\ S.R \rangle$	function definition elaboration
	— R can be E, D, or void

The context Γ *globalises* the scope of each variable. This is done to avoid complications that arise with the scope of local variables in the cases when a user-defined function is being unrolled. Furthermore, in the subset of Stan we defined, all variables are globally scoped, thus keeping the original scope of the variables is not necessary. Future work, however, might include extending the supported subset of Stan to include local variables, so that there is more flexibility, when using SlicStan, in deciding which variables are printed during inference.[2] This work will also have to take care of keeping information about variables' scope in the elaboration phase.

All elaboration rules make use of the context in a straightforward way — checking for overlapping variable names in the premise (e.g. $\Gamma_1 \cap \Gamma_2 = \varnothing$), and merging contexts in the conclusion ($\Gamma_1 \cup \Gamma_2$). The constraint $\Gamma_1 \cap \Gamma_2 = \varnothing$ can always be satisfied by α-converting statements and expressions using a suitable choice of local variable names. For simplicity, we ignore explanations about the α-conversion and the context Γ hereafter.

To demonstrate the meaning of the different elaboration pairs, it is best to focus on one of the user-defined-function-call elaboration rules. Take for example (ELAB ECALL). This rule takes a call to a user defined function $F_E(E_1, \ldots, E_N)$, which returns an expression E_F, and turns it into a sequence ot statements $S_{E_1}; a_1 = E'_1; \ldots; S_{E_N}; a_N = E'_N; S_F$, paired with the return expression E_F. Each statement S_{E_i} is the result statement of elaborating E_i, and $a_i = E'_i$ assigns the result expression E'_i to the argument a_i of the function F_E. Finally, S_F is the body of the function F_E. To illustrate this with a code example, suppose we have the following SlicStan program (ignoring variable declarations):

```
def my_normal(m, s){
  xr ~ normal(0, 1);
  return m + xr*s;
}
x = my_normal(10, 0.5);
```

Using (ELAB ECALL) and (ELAB ASSIGN) to elaborate `x = my_normal(10,0.5)`, we get:

```
m = 10;
s = 0.5;
xr ~ normal(0, 1);
x = m + xr*s;
```

[2]Stan does not save information about local variables. The output of running inference on a Stan program contains samples of the parameters, transformed parameters, and generated quantities only.

Elaboration Rules for Expressions:

(Elab Var)
$$\frac{}{x \Downarrow \langle \varnothing,\ Skip.x \rangle}$$

(Elab Const)
$$\frac{}{c \Downarrow \langle \varnothing,\ Skip.c \rangle}$$

(Elab Arr)
$$\frac{E_i \Downarrow \langle \Gamma_i,\ S_i.E_i' \rangle \quad \forall i \in 1..N \quad \bigcap_{i=1}^{N} \Gamma_i = \varnothing}{[E_1, \ldots, E_N] \Downarrow \langle \bigcup_{i=1}^{N} \Gamma_i,\ S_1; \ldots; S_N.([E_1', \ldots, E_N']) \rangle}$$

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

(Elab EPrim)
$$\frac{E_i \Downarrow \langle \Gamma_i,\ S_i.E_i' \rangle \quad \forall i \in 1..N \quad \bigcap_{i=1}^{N} \Gamma_i = \varnothing}{f(E_1, \ldots, E_N) \Downarrow \langle \bigcup_{i=1}^{N} \Gamma_i,\ S_1; \ldots; S_N.f(E_1', \ldots, E_N') \rangle}$$

(Elab ECall)
$$\frac{E_i \Downarrow \langle \Gamma_{E_i},\ S_{E_i}.E_i' \rangle \quad F_E \Downarrow \langle A_F, \Gamma_F,\ S_F.E_F \rangle \quad A_F = \{a_i : T_i \mid i \in 1..N\} \quad A_F \cap (\bigcap_{i=1}^{N} \Gamma_i) \cap \Gamma_F = \varnothing}{F_E(E_1, \ldots, E_N) \Downarrow \langle A_F \cup (\bigcup_{i=1}^{N} \Gamma_i) \cup \Gamma_F,\ S_{E_1}; a_1 = E_1'; \ldots; S_{E_N}; a_N = E_N'; S_F.E_F \rangle}$$

Elaboration Rules for L-Values:

(Elab LVal)
$$\frac{}{x \Downarrow \langle \varnothing,\ Skip.x \rangle}$$

(Elab LArrEl)
$$\frac{E \Downarrow \langle \Gamma,\ S.E' \rangle}{L[E] \Downarrow \langle \Gamma,\ S.(L[E']) \rangle}$$

The rest of the rules for expressions and l-values are straightforward — recursively elaborating each language construct, making sure there is no overlap between variable names by applying α-renaming when necessary, and combining the results.

(Elab DCall) and (Elab VCall) follow the pattern of (Elab ECall), but for calls of user-defined distributions and user-defined void functions respectively. (Elab Data) and (Elab Block) make sure to add the newly declared variable to the global scope, and otherwise remove the scope of that variable.

Elaboration Rules for Distributions:

(Elab DPrim)
$$\frac{E_i \Downarrow \langle \Gamma_i,\ S_i.E_i' \rangle \quad \forall i \in 1..N \quad \bigcap_{i=1}^{N} \Gamma_i = \varnothing}{f(E_1, \ldots, E_N) \Downarrow \langle \bigcup_{i=1}^{N} \Gamma_i,\ S_1; \ldots; S_N.f(E_1', \ldots, E_N') \rangle}$$

(Elab DCall)
$$\frac{E_i \Downarrow \langle \Gamma_{E_i},\ S_{E_i}.E_i' \rangle \quad F_D \Downarrow \langle A_F, \Gamma_F,\ S_F.D_F \rangle \quad A_F = \{a_i : T_i \mid i \in 1..N\} \quad A_F \cap (\bigcap_{i=1}^{N} \Gamma_i) \cap \Gamma_F = \varnothing}{F_D(E_1, \ldots, E_N) \Downarrow \langle A_F \cup (\bigcup_{i=1}^{N} \Gamma_i) \cup \Gamma_F,\ S_{E_1}; a_1 = E_1'; \ldots; S_{E_N}; a_N = E_N'; S_F.D_F \rangle}$$

Elaboration Rules for Statements:

$$\frac{E \Downarrow \langle \Gamma,\ S.E' \rangle \quad x \notin \Gamma}{x = E \Downarrow \langle \Gamma,\ S; x = E' \rangle}\ (\textsc{Elab Assign}) \qquad \frac{D \Downarrow \langle \Gamma,\ S.D' \rangle \quad x \notin \Gamma}{x \sim D \Downarrow \langle \Gamma,\ S; x \sim D' \rangle}\ (\textsc{Elab Model}) \qquad \frac{S \Downarrow \langle \Gamma,\ S' \rangle \quad x \notin \Gamma}{\{Tx; S\} \Downarrow \langle \{(T, x)\} \cup \Gamma,\ S' \rangle}\ (\textsc{Elab Block})$$

$$\frac{S \Downarrow \langle \Gamma,\ S' \rangle \quad x \notin \Gamma}{\textbf{data}\ \tau\, x; S \Downarrow \langle \{x : (\tau, \textsc{data})\} \cup \Gamma,\ S' \rangle}\ (\textsc{Elab Data})$$

$$\frac{S_1 \Downarrow \langle \Gamma_1,\ S_1' \rangle \quad S_2 \Downarrow \langle \Gamma_2,\ S_2' \rangle \quad \Gamma_1 \cap \Gamma_2 = \varnothing}{S_1; S_2 \Downarrow \langle \Gamma_1 \cup \Gamma_2,\ S_1'; S_2' \rangle}\ (\textsc{Elab Seq}) \qquad \frac{}{Skip \Downarrow \langle \varnothing,\ Skip \rangle}\ (\textsc{Elab Skip})$$

$$\frac{E_i \Downarrow \langle \Gamma_{E_i},\ S_{E_i}.E_i' \rangle \quad F_V \Downarrow \langle A_F, \Gamma_F,\ S_F \rangle \quad A_F = \{a_i : T_i \mid i \in 1..N\} \quad A_F \cap (\bigcap_{i=1}^{N} \Gamma_i) \cap \Gamma_F = \varnothing}{F_V(E_1, \ldots, E_N) \Downarrow \langle A_F \cup (\bigcup_{i=1}^{N} \Gamma_i) \cup \Gamma_F,\ S_{E_1}; a_1 = E_1'; \ldots; S_{E_N}; a_N = E_N'; S_F \rangle}\ (\textsc{Elab VCall})$$

The final set of elaboration rules is that of function definitions. The three rules (one for each type of function) follow a similar pattern — the body of the function, and the return value are elaborated, and their resulting contexts combined with the set of arguments.

Elaboration Rules for Function Definitions:

$$\frac{S \Downarrow \langle \Gamma_S,\ S' \rangle \quad A \cap \Gamma_S = \varnothing \quad A = \{a_1 : T_1, \ldots, a_N : T_N\}}{\textbf{def}\ F_V(T_1\, a_1, \ldots, T_N\, a_N)\ S \Downarrow \langle A, \Gamma_S,\ S' \rangle}\ (\textsc{Elab VDef})$$

$$\frac{S \Downarrow \langle \Gamma_S,\ S_S \rangle \quad E \Downarrow \langle \Gamma_E,\ S_E.E' \rangle \quad A \cap \Gamma_S \cap \Gamma_E = \varnothing \quad A = \{a_1 : T_1, \ldots, a_N : T_N\}}{\textbf{def}\ F_E(T_1\, a_1, \ldots, T_N\, a_N)\ S\ \textbf{return}\ E \Downarrow \langle A, \Gamma_S \cup \Gamma_E,\ S_S; S_E.E' \rangle}\ (\textsc{Elab EDef})$$

$$\frac{S \Downarrow \langle \Gamma_S,\ S_S \rangle \quad D \Downarrow \langle \Gamma_D,\ S_D.D' \rangle \quad A \cap \Gamma_S \cap \Gamma_D = \varnothing \quad A = \{a_1 : T_1, \ldots, a_N : T_N\}}{\textbf{def}\ F_D(T_1\, a_1, \ldots, T_N\, a_N)\ S\ \textbf{return}\ D \Downarrow \langle A, \Gamma_S \cup \Gamma_D,\ S_S; S_D.D' \rangle}\ (\textsc{Elab DDef})$$

Properties of the Elaboration Relations

Having defined the elaboration relation $\Downarrow$, we want to prove type preservation with respect to the declarative typing system from §§ 4.2.1.

Conjecture 2 (Type Preservation of $\Downarrow$)**.** *For all typing environments Γ and Γ', expressions E and E', l-values L and L', distributions D and D', statements S and S', and level types ℓ:*

(1) *If $\Gamma \vdash E : (\tau, \ell)$ and $E \Downarrow \langle \Gamma',\ S.E' \rangle$, then $\Gamma, \Gamma' \vdash E' : (\tau, \ell)$ and $\Gamma, \Gamma' \vdash S : \ell$*

(2) *If* $\Gamma \vdash L : (\tau, \ell)$ *and* $L \Downarrow \langle \Gamma',\ S.L' \rangle$, *then* $\Gamma, \Gamma' \vdash L' : (\tau, \ell)$ *and* $\Gamma, \Gamma' \vdash S : \ell$

(3) *If* $\Gamma \vdash D : (\tau, \ell)$ *and* $D \Downarrow \langle \Gamma',\ S.D' \rangle$, *then* $\Gamma, \Gamma' \vdash D' : (\tau, \ell)$ *and* $\Gamma, \Gamma' \vdash S : \ell$

(4) *If* $\Gamma \vdash S : \ell$ *and* $S \Downarrow \langle \Gamma',\ S' \rangle$, *then* $\Gamma, \Gamma' \vdash S' : \ell$.

We leave proving Conjecture 2 by structural induction for future work.

4.3.3 Transformation

Transformation is the last step taken in the translation of SlicStan to Stan. This step takes a pair of a context Γ and an *elaborated* (free of calls to user-defined functions, blocks, and data declarations) SlicStan statement S. This pair is then transformed to a Stan program P, where P is of the form given in §§ 4.3.1:

$$
\begin{aligned}
P = \ & \textbf{data}\{\ \Gamma_d\ \} \\
& \textbf{transformed data}\{\ \Gamma'_d\ \ S'_d\ \} \\
& \textbf{parameters}\{\ \Gamma_m\ \} \\
& \textbf{transformed parameters}\{\ \Gamma'_m\ \ S'_m\ \} \\
& \textbf{model}\{\ \Gamma''_m\ \ S''_m\ \} \\
& \textbf{generated quantities}\{\ \Gamma_q\ \ S_q\ \}
\end{aligned}
$$

This is done in two steps: (1) transformation of the context Γ to a Stan program P_1 containing declarations of the variables in Γ ($\Gamma \Downarrow_S P_1$), and (2) transformation of the elaborated SlicStan program S to a Stan program P_2 ($S \Downarrow_\Gamma P_2$).

Transformation Relations

$\Gamma \Downarrow_S P$	variable declarations transformation
$S \Downarrow_\Gamma P$	statement transformation
$\langle \Gamma,\ S \rangle \Downarrow_T P$	top-level transformation

We combine (see definition Definition 4) the resulting programs of the previous two to get the final Stan program. In other words, given a pair $\langle \Gamma,\ S \rangle$, we use the (Trans Pair) rule as follows:

$$
\text{(Trans Pair)}\quad \frac{\Gamma \Downarrow_S P_1 \quad S \Downarrow_\Gamma P_2}{\langle \Gamma,\ S \rangle \Downarrow_T P_1; P_2}
$$

The rest of this subsection presents the transformation rules associated with the relations $\Downarrow_S$ and $\Downarrow_\Gamma$. In the cases where particular blocks are not mentioned, the missing blocks are considered empty. In the cases where only a typing environment/statement is mentioned for some block that contains both, the one missing is considered empty/*Skip*. In other

words, by writing, for example, **transformed data**$\{x : (\tau, \text{DATA})\}$, we refer to the full Stan program:

$$\begin{aligned} P = \ & \textbf{data}\{\} \ \ \textbf{transformed data}\{x : (\tau, \text{DATA}) \ \ \mathit{Skip}\} \\ & \textbf{parameters}\{\} \ \ \textbf{transformed parameters}\{\} \\ & \textbf{model}\{\} \ \ \textbf{generated quantities}\{\} \end{aligned}$$

The transformation rules for declarations take a SlicStan typing environment Γ and split it into the typing environments for each Stan block. Variables are being "assigned" to different blocks based on their level, and whether or not they appear on the left hand side of an assignment.

Transformation Rules for Declarations:

(Trans Data)
$$\frac{\Gamma \Downarrow_S P \qquad x \notin A(S)}{\Gamma, x : (\tau, \text{DATA}) \Downarrow_S \textbf{data}\{x : (\tau, \text{DATA})\}; P}$$

(Trans TrData)
$$\frac{\Gamma \Downarrow_S P \qquad x \in A(S)}{\Gamma, x : (\tau, \text{DATA}) \Downarrow_S \textbf{transformed data}\{x : (\tau, \text{DATA})\}; P}$$

(Trans Param)
$$\frac{\Gamma \Downarrow_S P \qquad x \notin A(S)}{\Gamma, x : (\tau, \text{MODEL}) \Downarrow_S \textbf{parameters}\{x : (\tau, \text{MODEL})\}; P}$$

(Trans TrParam)
$$\frac{\Gamma \Downarrow_S P \qquad x \in A(S)}{\Gamma, x : (\tau, \text{MODEL}) \Downarrow_S \textbf{transformed parameters}\{x : (\tau, \text{MODEL})\}; P}$$

(Trans GenQuant)
$$\frac{\Gamma \Downarrow_S P}{\Gamma, x : (\tau, \text{GENQUANT}) \Downarrow_S \textbf{generated quantities}\{x : (\tau, \text{GENQUANT})\}; P}$$

(Trans Empty)
$$\frac{}{\varnothing \Downarrow_\Gamma \varnothing}$$

Finally, the transformation rules for statements take an elaborated SlicStan statement S, and split it into the statements associated with each Stan block. Assignments are placed in **transformed data**, **transformed parameters**, or **generated quantities**, based on the level of the left-hand side of the assignment ((Trans AssignData), (Trans AssignParam),and (Trans AssignGenQuant)). All sampling statements are placed in the **model** block (Trans Model).

Transformation Rules for Statements:

$$\frac{\Gamma \vdash L \to (\tau, \text{DATA})}{L = E \Downarrow_\Gamma \textbf{transformed data}\{L = E\}} \text{(Trans AssignData)} \qquad \frac{\Gamma \vdash L \to (\tau, \text{MODEL})}{L = E \Downarrow_\Gamma \textbf{transformed parameters}\{L = E\}} \text{(Trans AssignParam)}$$

$$\frac{\Gamma \vdash L \to (\tau, \text{GENQUANT})}{L = E \Downarrow_\Gamma \textbf{generated quantities}\{L = E\}} \text{(Trans AssignGenQuant)} \qquad \frac{}{x \sim D \Downarrow_\Gamma \textbf{model}\{x \sim D\}} \text{(Trans Model)}$$

$$\frac{S_1 \Downarrow_\Gamma P_1 \quad S_2 \Downarrow_\Gamma P_2}{S_1; S_2 \Downarrow_\Gamma P_1; P_2} \text{(Trans Seq)} \qquad \frac{}{Skip \Downarrow_\Gamma \varnothing} \text{(Trans Skip)}$$

Expressions, l-values and distributions are not transformed, as in an elaborated SlicStan statement they are equivalent to Stan's expressions, l-values and distributions.

Definition 4. *Sequence of Stan programs $P_1; P_2$*
Let P_1 and P_2 be two Stan programs, such that for $i = 1, 2$:

$$\begin{aligned} P_i = \ & \textbf{data}\{\, \Gamma_d^{(i)} \,\} \\ & \textbf{transformed data}\{\, \Gamma_d^{(i)\prime} \quad S_d^{(i)\prime} \,\} \\ & \textbf{parameters}\{\, \Gamma_m^{(i)} \,\} \\ & \textbf{transformed parameters}\{\, \Gamma_m^{(i)\prime} \quad S_m^{(i)\prime} \,\} \\ & \textbf{model}\{\, \Gamma_m^{(i)\prime\prime} \quad S_m^{(i)\prime\prime} \,\} \\ & \textbf{generated quantities}\{\, \Gamma_q^{(i)} \quad S_q^{(i)} \,\} \end{aligned}$$

The sequence of P_1 and P_2, written $P_1; P_2$ is then defined as:

$$\begin{aligned} P_i = \ & \textbf{data}\{\, \Gamma_d^{(1)}, \Gamma_d^{(2)} \,\} \\ & \textbf{transformed data}\{\, \Gamma_d^{(1)\prime}, \Gamma_d^{(2)\prime} \quad S_d^{(1)\prime}; S_d^{(2)\prime} \,\} \\ & \textbf{parameters}\{\, \Gamma_m^{(1)}, \Gamma_m^{(2)} \,\} \\ & \textbf{transformed parameters}\{\, \Gamma_m^{(1)\prime}, \Gamma_m^{(2)\prime} \quad S_m^{(1)\prime}; S_m^{(2)\prime} \,\} \\ & \textbf{model}\{\, \Gamma_m^{(1)\prime\prime}, \Gamma_m^{(2)\prime\prime} \quad S_m^{(1)\prime\prime}; S_m^{(2)\prime\prime} \,\} \\ & \textbf{generated quantities}\{\, \Gamma_q^{(1)}, \Gamma_q^{(2)} \quad S_q^{(1)}; S_q^{(2)} \,\} \end{aligned}$$

Properties of the Transformation Relations

Having defined the transformation relation $\Downarrow_T$, we want to prove type preservation with respect to the declarative typing system from §§ 4.2.1.

Conjecture 3 (Type Preservation of $\Downarrow_T$)**.** *For all typing environments* Γ*, statements* S*, where* S *does not contain any blocks or calls to user-defined functions, level types* ℓ*, and Stan programs* P*:*

$$\textit{If } \Gamma \vdash S : \ell \textit{ and } \langle \Gamma,\ S \rangle \Downarrow_T P, \textit{ then } \vdash_{Stan} P$$

We leave proving Conjecture 3 by structural induction for future work.

4.4 Implementation

This section briefly outlines how the rules described so far were implemented to produce the SlicStan compiler, which takes as input a plain text SlicStan source file (`.slic`), and outputs a Stan source code file (`.stan`). All components of the compiler were implemented in F#, which was chosen for its algebraic datatypes and functional syntax that allow for straightforward deep embedding of SlicStan into F#.

Parsing

The SlicStan lexer and parser were generated using the FsLexYacc library[3] — a library for Lex/Yacc family lexer/parser generators tools for F#. The parser takes as an input a plain text source file of a SlicStan program, and (provided that no syntax error occurs) generates an abstract syntax tree (AST), which is a well-typed F# expression. For each variable declaration in the source file, if a level type for the variable is not specified, the parser generates a unique level type placeholder for this variable.

Type Checking and Type Inference

After the AST has been generated, it is passed to the typecheck function, which applies the rules from §§ 4.2.2 to typecheck the program, producing constraints where the level types are unknown. It then follows the procedure described in §§ 4.2.3 to resolve those constraints, and returns a SlicStan syntax tree expression, where the level type placeholders are replaced by their respective inferred levels.

Translation

The, now fully typed, SlicStan AST is passed to the elaborate function, which applies the rules described in §§ 4.3.2 elaborating the program to a user-defined-function-call-free statement. The result is pipelined to the translate function, which in turn applies the rules from §§ 4.3.3, returning a Stan AST.

Finally, the Stan AST is pretty-printed to a `.stan` source file, which can be compiled and ran using any of the supported Stan interfaces.[4]

[3] FsLexYacc: http://fsprojects.github.io/FsLexYacc/

[4] Compilation has been tested only using PyStan — the Python interface to Stan — however Stan programs are portable across interfaces: http://mc-stan.org/users/interfaces/

4.5 Design of Extensions

So far this chapter has focused on the aspects of SlicStan that have been implemented and tested. Additional features have been considered, the preliminary design of which we present here. Those features have not been implemented, and are not part of the language that is discussed in this dissertation.

4.5.1 Direct Access to the Log Probability Distribution

One functionality that Stan provides is incrementing the accumulated **target** log probability density directly. As previously mentioned in §§ 2.1.1, writing `x ~ normal(m, s)` in Stan is equivalent to writing **target** `+= normal_lpdf(x | m, s)`. This syntax can be very convenient when a *Jacobian adjustment* needs to be made, due to a non-linear change of variables.[5]

Suppose, for example, that a variable X has density $p_X(x)$. We are interested in finding the density $p_Y(y)$ of a variable $Y = f(X)$, where f is invertible. To preserve probability mass, we want:

$$p_X(x)\,|dx| = p_Y(y)\,|dy|$$

for small corresponding dx and dy. This gives us:

$$\begin{aligned} p_Y(y) &= p_X(x)\left|\frac{dx}{dy}\right| \\ &= p_X\left(f^{-1}(y)\right)\left|\frac{d}{dy}f^{-1}(y)\right| \end{aligned}$$

If we want to implement the above in Stan, we might write $f^{-1}(y) \sim p_X$, which will be equivalent to **target** `+=` $\log p_X(f^{-1}(y))$. We still need to apply the Jacobian adjustment by directly adding it to the log probability function: **target** `+=` $\log\left|\frac{d}{dy}f^{-1}(y)\right|$.

Adding a way to directly access the **target** variable in SlicStan will make it possible to perform non-linear change of variables. The language can be extended to support this, by adding a new statement allowing access to the log probability density, and implementing a few additional typing and translation rules:

$$\frac{\Gamma \vdash E : (\tau, \text{MODEL})}{\Gamma \vdash \textbf{target } \texttt{+=}E : \text{MODEL}} \quad (\textsc{LogProb})$$

$$\frac{E \Downarrow \langle \Gamma,\ S.E' \rangle}{\textbf{target } \texttt{+=}E \Downarrow \langle \Gamma,\ S; \textbf{target } \texttt{+=}E' \rangle} \quad (\textsc{Elab LogProb})$$

$$\frac{}{\textbf{target } \texttt{+=}E \Downarrow_\Gamma \textbf{model}\{\textbf{target } \texttt{+=}E\}} \quad (\textsc{Trans LogProb})$$

[5] Note that in the reparameterisation examples shown previously, the change of variables is linear (e.g. $y = \sigma x + \mu$, were $x \sim \mathcal{N}(0, 1)$), thus the Jacobian is constant with respect to the parameters, and adjustment is not needed.

4.5.2 User-defined Functions Vectorisation

One powerful feature of Stan, both in terms of usability and efficiency, is that of vectorised expressions. As outlined in §§ 2.3.3, Stan's expressions can be vectorised (similarly to NumPy's *broadcasting*[6]), so that writing `x ~ normal(0,1);`, where `x` is a vector of size `N`, is semantically equivalent to `for (n in 1:N){x[n] ~ normal(0,1);}`.

Currently, an attempt to vectorise a call to a user-defined function in SlicStan will type-check, but the translation rules (§ 4.3) lack the necessary type context to implement this vectorisation correctly. For example, consider the user-defined function `my_normal` that implements reparameterisation of a variable drawn from a standard Gaussian distribution:

```
def my_normal(real m, real s){
  real xr ~ normal(0, 1);
  return m + s*xr;
}
```

If we attempt to elaborate the statement `vector[N] x ~ my_normal(10, 0.5)`, we get:

```
real m = 10;
real s = 0.5;
real xr ~ normal(0, 1);
vector[N] x = m + s*xr;
```

This will result in an illegal Stan program, as the expression `x = m + s*xr;` has a vector on the left-hand side, and a scalar on the right-hand side. Even if such statement can be vectorised in Stan, we will still be transforming the same `xr` parameter for each `x[n]`, whereas what we probably intend when writing `vector[N] x ~ my_normal(10, 0.5)`, is:

```
real m = 10;
real s = 0.5;
vector[N] xr ~ normal(0, 1);
vector[N] x = m + s*xr;
```

We consider extending the elaboration rules of SlicStan, so that vectorisation of user-defined function can be implemented. For example, the elaboration relation $\Downarrow$, can be extended to carry a type context Γ (necessary in order to check if a vectorised call is being made, and to obtain array sizes), and a "size" of the vectorisation N:

Extended Elaboration Relation

$\Gamma \vdash E \Downarrow_N \langle \Gamma',\ S.E' \rangle$	expression elaboration
$\Gamma \vdash L \Downarrow_N \langle \Gamma',\ S.L' \rangle$	l-value elaboration
$\Gamma \vdash D \Downarrow_N \langle \Gamma',\ S.D' \rangle$	distribution elaboration
$\Gamma \vdash S \Downarrow_N \langle \Gamma',\ S' \rangle$	statement elaboration
$\Gamma \vdash F \Downarrow_N \langle \Gamma',\ S.R \rangle$	function definition elaboration
	— R can be E, D, or void

[6] See https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

4.5.3 Mixture Models

Chapter 3 listed a few possible directions of work on the Stan modelling language, one of which was about *mixture models* — models where a variable x can be drawn from one of several different distributions. For example, suppose that we have two Gaussian distributions, both of a unit variance, and for each $i = 1, 2$, the i^{th} distribution is centred at some μ_i. Suppose also that x is drawn from the i^{th} such Gaussian:

$$p(x \mid i) = \mathcal{N}(x \mid \mu_i, 1)$$

If i — the label of the Gaussian that x was drawn from — comes from a $Bernoulli(\pi)$ distribution, the join probability density of x and i is:

$$p(x, i) = p(x \mid i)p(i)$$

Sampling directly from such a density is not possible in Stan, as it is not differentiable with respect to the *discrete* parameter i. However, we can sample from the *marginal* distribution of x:

$$\begin{aligned} p(x) &= p(x \mid i = 1)p(i = 1) + p(x \mid i = 2)p(i = 2) \\ &= \pi\,\mathcal{N}(x \mid \mu_1, 1) + (1 - \pi)\,\mathcal{N}(x \mid \mu_2, 1) \end{aligned}$$

To write this in Stan, we need to directly access the **target** probability density, incrementing it with $\log p(x)$:

```
target += log( pi   * exp(normal_lpfd(x | mu1, 1))
          + (1-pi) * exp(normal_lpfd(x | mu2, 1)) )
```

Or using the numerically stable log_mix function:

```
target += log_mix( pi, normal_lpfd(x | mu1, 1),
                       normal_lpfd(x | mu2, 1) );
```

We suggest the introduction of additional language constructs to allow for *mixture models* to be expressed more naturally. For example, in such binary cases, SlicStan can be extended to support an **if–then–else** statement, where the conditioning can be with respect to a (boolean) distribution:

$S ::= \ldots \mid \textbf{if}(D)\ S_1\ \textbf{else}\ S_2$	probabilistic **if** statement

Stan supports **if** statements, however they can depend only on an expression E. Extending SlicStan with syntax for probabilistic **if** statements allows us to write the program:

```
data vector[2] mu;
data real pi;
real x;

if(bernoulli(pi)) x ~ normal(mu[1], 1);
else x ~ normal(mu[2], 1);
```

And translate to Stan as follows:

```
data {
  vector[2] mu;
  real pi;
}
parameters {
  real x;
}
model {
  real target0;
  real logprob1;
  real logprob2;

  // copy of the log likelihood at start
  target0 = target();

  // run first branch, which increments target
  x ~ normal(mu[1], 1);
  logprob1 = target() - target0;
  target0 = target();

  // run second branch, which increments target
  x ~ normal(mu[2], 1);
  logprob2 = target() - target0;

  target += log_mix (pi, logprob1, logprob2) - logprob1 - logprob2;
}
```

We tested this usage of the **target** variable to work as expected, and can be extended with **case**(i ~ D)S syntax, with D being a discrete distribution. However, we have not measured how the manipulation of the **target** in this way impacts performance. Further studying of the problem is required to determine if such an approach will be useful in practice, and if more efficient solutions exist.

Chapter 5

Demonstration and Analysis

In this chapter, we demonstrate and discuss the functionality of SlicStan. We compare several Stan code examples, taken from Stan's Reference Manual [Stan Development Team, 2017] and Stan's GitHub repositories,[1] with their equivalent written in SlicStan, and analyse the differences.

The focus is on three comparison points — abstraction, code refactoring, and code reuse. We firstly show that SlicStan allows for a more powerful abstraction of the underlying probabilistic model, due to the absence of program blocks (§ 5.1). Secondly, we demonstrate the advantages of the more compositional syntax, when code refactoring is needed (§ 5.2). The last comparison point shows the usage of more flexible user-defined functions, and points out a few limitations of SlicStan (§ 5.3).

In the last section of this chapter (§ 5.4), we also report results on how SlicStan's compiler performance scales with the size of the program.

5.1 Abstraction

This section works through two examples to demonstrate one of the core ideas behind SlicStan — the lack of program blocks. We claim that SlicStan's blockless syntax provides a better abstraction for the underlying probabilistic models, as it does not require the programmer to know the specifics of the Hamiltonian Monte Carlo sampling algorithm.

5.1.1 Opening Example

In Chapter 3, we showed a simple probabilistic model written in both SlicStan and Stan. We revisit this example overleaf, to compare more closely the syntax of the two languages. On the left is that same piloting example written in SlicStan, and on the right — in Stan. Note that translating this SlicStan snippet results in precisely the same Stan program, as the hand-optimised program taken from the manual (up to ordering of declarations within blocks), and we therefore present only the latter here.

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

Looking at the code, one thing is immediately evident: SlicStan's program is shorter than that of Stan. This is not an isolated example — SlicStan's expressions and statements are very similar (identical if no user-defined functions are present) to those of Stan. However, unlike Stan, there are no block definitions, and more lines can be combined. In particular, parameter declarations can appear together with a modelling statement ($\sim$) associated with them (lines 8 and 10 of SlicStan versus lines 12, 13, 19 and 20 in Stan).

Moreover, the lack of blocks in SlicStan, makes it more flexible in terms of order of statements. The code here is written to be close to the original Stan code, with only the definition of sigma_y coming in-between the modelling statements for mu_y and y to demonstrate that flexibility. This particular model, however, can be expressed in SlicStan by using any ordering on the statements, as long as variables are not used before they are declared. In Stan this is not always possible, and might result in closely related statements being far away from each other in the code (e.g. the definition of sigma_y in line 16, and its use to model y in line 21).

SlicStan

```
1  data int N;
2  data real[N] y;
3  data real mu_mu;
4  data real sigma_mu;
5
6  real alpha = 0.1;
7  real beta = 0.1;
8  real tau_y ~ gamma(alpha, beta);
9
10 real mu_y ~ normal(0, 1);
11
12 real sigma_y = pow(tau_y, -0.5);
13 y ~ normal(mu_y, sigma_y);
14
15 real variance_y = pow(sigma_y, 2);
```

Stan

```
1  data {
2    real mu_mu;
3    real sigma_mu;
4    int N;
5    real y[N];
6  }
7  transformed data {
8    real alpha = 0.1;
9    real beta = 0.1;
10 }
11 parameters {
12   real mu_y;
13   real tau_y;
14 }
15 transformed parameters {
16   real sigma_y = pow(tau_y,-0.5);
17 }
18 model {
19   tau_y ~ gamma(alpha, beta);
20   mu_y ~ normal(0, 1);
21   y ~ normal(mu_y, sigma_y);
22 }
23 generated quantities {
24   real variance_y = pow(sigma_y,2);
25 }
```

Example adapted from [Stan Development Team, 2017, p. 101].

An important observation is that with SlicStan there is no need to understand where and how different statements are executed in order to perform inference. The SlicStan code is translated exactly 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 program that does not compile. And 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 (semantically) the same model, however it won't be optimised (see § 2.3).

5.1.2 Seeds

To demonstrate what was discussed above with one more example, we take the "Seeds" example introduced by Lunn et al. [2012, p. 300] in "*The BUGS Book*". In this example, we have I plates, with plate i having a total of N_i seeds on it, n_i of which have germinated. Moreover, each plate i has one of 2 types of seeds $x_1^{(i)}$, and one of 2 types of root extract $x_2^{(i)}$. We are interested in modelling the number of germinated seeds based on the type of seed and root extract, which we do in two steps. Firstly, we model the number of germinated seeds with a Binomial distribution, whose success probability is the probability of a single seed germinating:

$$n_i \sim Binomial(N, p_i)$$

Next, we model the probability of a single seed on plate i germinating as the output of a logistic regression with input variables the type of seed and root extract:

$$p_i = \sigma(\alpha_0 + \alpha_1 x_1^{(i)} + \alpha_2 x_2^{(i)} + \alpha_{12} x_1^{(i)} x_2^{(i)} + \beta^{(i)})$$

In the above, $\alpha_0, \alpha_1, \alpha_2, \alpha_{12}$ and $\beta^{(i)}$ are parameters of the model, with $\beta^{(i)}$ allowing for *over-dispersion* (see §§ 5.2.2).

The next page shows the "Seeds" model written in SlicStan (left) and in Stan (right). The Stan code was adapted from the example models listed on Stan's GitHub page. The SlicStan version translates to an identical (up to ordering within blocks) Stan program, therefore only the original Stan program is shown here.

As before, we see that SlicStan's code is shorter than that of Stan. It also allows for more flexibility in the order of declarations and definitions, making it possible to keep related statements together (e.g. lines 14 and 15 of the example written in SlicStan). Once again, SlicStan provides more abstraction, as the programmer does not have to specify how each variable of the model should be treated by the underlying inference algorithm. Instead it automatically determines this when it translates the program to Stan.

"Seeds" in SlicStan

```
data int I;
data int[I] n;
data int[I] N;
data vector[I] x1;
data vector[I] x2;

vector[I] x1x2 = x1 .* x2;

real alpha0 ~ normal(0.0,1000);
real alpha1 ~ normal(0.0,1000);
real alpha2 ~ normal(0.0,1000);
real alpha12 ~ normal(0.0,1000);

real tau ~ gamma(0.001,0.001);
real sigma = 1.0 / sqrt(tau);

vector[I] b ~ normal(0.0, sigma);
n ~ binomial_logit[2](N, alpha0
                      + alpha1 * x1
                      + alpha2 * x2
                      + alpha12 * x1x2
                      + b);
```

"Seeds" in Stan

```
data {
  int I;
  int n[I];
  int N[I];
  vector[I] x1;
  vector[I] x2;
}

transformed data {
  vector[I] x1x2 = x1 .* x2;
}
parameters {
  real alpha0;
  real alpha1;
  real alpha12;
  real alpha2;
  real tau;
  vector[I] b;
}
transformed parameters {
  real sigma = 1.0 / sqrt(tau);
}
model {
  alpha0 ~ normal(0.0,1000);
  alpha1 ~ normal(0.0,1000);
  alpha2 ~ normal(0.0,1000);
  alpha12 ~ normal(0.0,1000);
  tau ~ gamma(0.001,0.001);

  b ~ normal(0.0, sigma);
  n ~ binomial_logit[2](N, alpha0
                        + alpha1 * x1
                        + alpha2 * x2
                        + alpha12 * x1x2
                        + b);
}
```

Example adapted from https://github.com/stan-dev/example-models/.

[2]Stan's binomial_logit distribution is a numerically stable way to use a logistic sigmoid in combination with a Binomial distribution.

5.2 Code Refactoring

This section demonstrates with two examples how the removal of program blocks can lead to a an easier to refactor code. In both cases we start from a simpler model, motivate the changes to be done, and describe what the refactored program looks like.

5.2.1 Regression with and without Measurement Error

We begin with another example taken from Stan's Reference Manual: linear regression with and without accounting for measurement error. The initial model we have is a simple Bayesian linear regression with N predictor points $\mathbf{x}$, and N outcomes $\mathbf{y}$. The model has 3 parameters — the intercept α, the slope β, and the amount of noise σ. We then have:

$$\mathbf{y} \sim \mathcal{N}(\alpha\mathbf{1} + \beta\mathbf{x}, \sigma\mathbb{I})$$

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

$$\mathbf{x}_{meas} \sim \mathcal{N}(\mathbf{x}, \tau\mathbb{I})$$

The next page shows those two models written in SlicStan (left) and Stan (right). Ignoring all the lines/corrections in red gives us the initial regression model — the one that *does not* account for measurement errors. The entire code, including the red corrections, gives us the second regression model — the one that *does* account for measurement errors. We see that 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).

We see that performing this refactoring requires the same amount of code in SlicStan and Stan. However, in SlicStan these changes interfere much less with the code already

written — we can add the statements that extend our model anywhere we like in the code (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 original aim of making Stan more compositional — composing separate programs/adding program extensions is easier in SlicStan.

Regression in SlicStan

```
data int N;
data vector[N] x;
data vector[N] y;

real alpha ~ normal(0, 10);
real beta ~ normal(0, 10);
real sigma ~ cauchy(0, 5);

y ~ normal(alpha + beta*x, sigma);

real mu_x;
real sigma_x;
x ~ normal(mu_x, sigma_x);

data real tau;
data vector[N] x_meas
          ~ normal(x, tau);
```

Regression in SlicStan

```
data {
  int N;
  vector[N] x_meas;
  real tau;
  vector[N] x;
  vector[N] y;
}
parameters {
  vector[N] x;
  real mu_x;
  real sigma_x;

  real alpha;
  real beta;
  real sigma;
}
model {
  x ~ normal(mu_x, sigma_x);
  x_meas ~ normal(x, tau);
  y ~ normal(alpha + beta*x, sigma);

  alpha ~ normal(0, 10);
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);
}
```

Example adapted from [Stan Development Team, 2017, p. 202].

5.2.2 Cockroaches

We conclude the demonstration of SlicStan's features with one more example comparing the process of code refactoring in SlicStan and Stan. The "Cockroaches" example is described by Gelman and Hill [2007, p. 161], and it concerns measuring the effects of integrated pest management on reducing cockroach numbers in apartment blocks. They use *Poisson regression* to model the number of caught cockroaches y_i in a single apartment i, with exposure u_i (the number of days that the apartment had cockroach traps in it), and regression predictors:

- the pre-treatment cockroach level r_i;
- whether the apartment is in a senior building (restricted to the elderly), s_i; and
- the treatment indicator t_i.

In other words, with $\beta_0, \beta_1, \beta_2, \beta_3$ being the regression parameters, we have:

$$y_i \sim Poisson(u_i \exp(\beta_0 + \beta_1 r_i + \beta_2 s_i + \beta_3 t_i))$$

After specifying their model in the way described above, Gelman and Hill simulate a replicated dataset $\mathbf{y}_{rep}$, and compare it to the actual data $\mathbf{y}$ to find that the variance of the simulated dataset is much lower than that of the real dataset. In statistics, this is called *overdispersion*, and is often encountered when fitting models based on a single parameter distributions,[3] such as the Poisson distribution. A better model for this data would be one that includes an *overdispersion parameter* $\boldsymbol{\lambda}$ that can account for the greater variance in the data.

The next page shows the "Cockroach" example before and after adding the overdispersion parameter, in both SlicStan (left) and Stan (right). Like in the previous subsection, ignoring the lines in red gives us the first model (the one that does not account for overdispersion), while adding them gives us the second model (the one with the additional parameter $\boldsymbol{\lambda}$). Once again, we see that modifying the model to account for overdispersion requires adding a similar number of statements in both languages. However, those statements can be added anywhere in the program (as long as it's before the statement modelling y, which depends on $\boldsymbol{\lambda}$). Stan, on the other hand, introduces an entire new to this program block — **transformed parameters** — to calculate the deviation σ from the precision hyperparameter τ.

[3] In a distribution specified by a single parameter α, the mean and variance both depend on α, and are therefore not independent.

"Cockroaches" in SlicStan

```
data int N;
data vector[N] exposure2;
data vector[N] roach1;
data vector[N] senior;
data vector[N] treatment;

vector[N] log_expo = log(exposure2);

vector[4] beta;

real tau ~ gamma(0.001, 0.001);
real sigma = 1.0 / sqrt(tau);
vector[N] lambda ~ normal(0, sigma);

data int[N] y
    ~ poisson_log[4](log_expo + beta[1]
            + beta[2] * roach1
            + beta[3] * treatment
            + beta[4] * senior
            + lambda);
```

"Cockroaches" in Stan

```
data {
  int N;
  vector[N] exposure2;
  vector[N] roach1;
  vector[N] senior;
  vector[N] treatment;
  int y[N];
}
transformed data {
  vector[N] log_expo = log(exposure2);
}
parameters {
  vector[4] beta;
  vector[N] lambda;
  real tau;
}
transformed parameters {
  real sigma = 1.0 / sqrt(tau);
}
model {
  tau ~ gamma(0.001, 0.001);
  lambda ~ normal(0, sigma);
  y ~ poisson_log[4](log_expo + beta[1]
            + beta[2] * roach1
            + beta[3] * treatment
            + beta[4] * senior
            + lambda);
}
```

Example adapted from https://github.com/stan-dev/example-models/.

5.3 Code Reuse

In this section, we demonstrate the usage of more flexible user-defined functions in SlicStan, which allow for better code reuse, and therefore can lead to shorter, more readable code. The examples to follow focus on one particular kind of operation important to hierarchical models and MCMC inference — model reparameterisation. This concept was already outlined in §§ 2.3.2, but here we describe it in more detail, and show how SlicStan addresses it.

[4]Stan's poisson_log function is a numerically stable way to model a Poisson variable where the Poisson event rate is of the form e^{α} for some α.

Reparameterising a model means expressing it in terms of different parameters, so that the original parameters can be recovered from the new ones. For example, in the Neal's Funnel model described earlier (§§ 2.3.2), the model parameters are changed from $\mathbf{x}$ and y to the standard normal parameters $\mathbf{x_{raw}}$ and y_{raw}, and the original parameters are recovered using shifting and scaling. This reparameterisation is a special case of a more general transform introduced by Papaspiliopoulos et al. [2007].

In Stan, centred parameterisation is more natural to write, and also more interpretable. However, as the Neal's Funnel example demonstrates, in some cases, especially when there is little data available, using non-centred parameterisation could be vital to the performance of the inference algorithm. The centred to non-centred parameterisation transformation is therefore common to Stan models, and is extensively described in Stan's Manual as a useful technique.

The code example below shows how a (centred) Gaussian variable $x \sim \mathcal{N}(\mu, \sigma)$ is parametrised in Stan (right). As this transformation involves introducing a new parameter x_{raw}, this cannot be done by defining a function in Stan. SlicStan, on the other hand, introduces more flexible user-defined functions, which are not constrained by the presence of program blocks. Reparameterising the model to a non-centred parameterisation can then be implemented by simply calling the function my_normal (left).

"Non-centred Parameterisation" in SlicStan

```
1  def my_normal(real m, real s){
2    real x_raw ~ normal(0, 1);
3    return s * x_raw + m;
4  }
5
6  real mu;
7  real sigma;
8  real x = my_normal(mu, sigma);
```

"Non-centred Parameterisation" in Stan

```
1  parameters {
2    real mu;
3    real sigma;
4    real x_raw;
5  }
6  transformed parameters {
7    real x = sigma * x_raw + mu;
8  }
9  model {
10   x_raw ~ normal(0, 1);
11 }
```

5.3.1 Neal's Funnel

Next, we demonstrate how this applies to the Neal's Funnel model. Using the reparameterisation function my_normal twice, the non-centred SlicStan program (left) is only longer than its centred version (**real** y $\sim \mathcal{N}(0,3)$; **real** x $\sim \mathcal{N}(0, e^{y/2})$), due to the presence of the definition of the function. In comparison, Stan requires defining the new parameters x_{raw} and y_{raw} (lines 2,3), moving the declarations of x and y to the **transformed parameters** block (lines 6,7), defining them in terms of the parameters (lines 8,9), and changing the definition of the join density accordingly (lines 12,13).

"Neal's Funnel" in SlicStan

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

"Neal's Funnel" in Stan

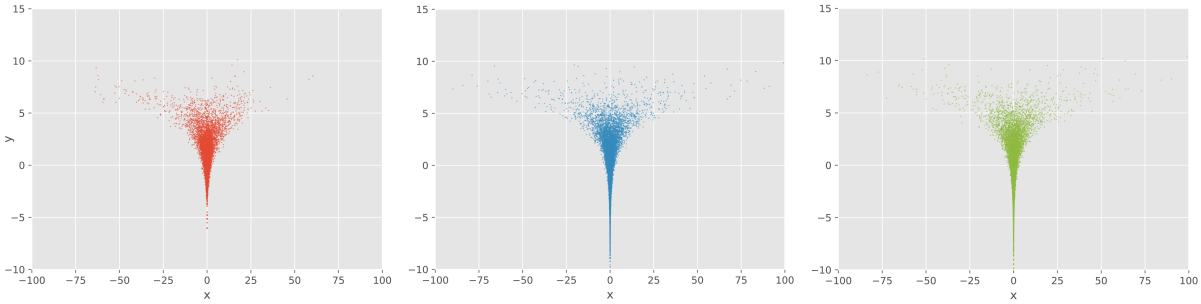
```
parameters {
  real y_raw;
  real x_raw;
}
transformed parameters {
  real y;
  real x;
  y = 3.0 * y_raw;
  x = exp(y/2) * x_raw;
}
model {
  y_raw ~ normal(0, 1);
  x_raw ~ normal(0, 1);
}
```

Example adapted from [Stan Development Team, 2017, p. 338].

Unlike the previous example, SlicStan translates to a different program than the original Stan program given in the manual. We present the "translated" Neal's Funnel model, as outputted by the SlicStan compiler on the next page. We notice a major difference between the two Stan programs — in one case the variables of interest x and y are defined in the **transformed parameters** block, while in the other they are defined in the **generated quantities** block. In an intuitive, centred parameterisation of this model, x and y are in fact the parameters. Therefore, it is much more natural to think of those variables as transformed parameters when using a non-centred parameterisation. However, as previously described in § 2.3, variables declared in the **transformed parameters** block are re-evaluated at every leapfrog, while those declared in the **generated quantities** block are re-evaluated at every sample. This means that even though it's more intuitive to think of x and y as transformed parameters (original Stan program), declaring them as generated quantities where possible results in a better optimised inference algorithm in the general case. As SlicStan assigns variables to different blocks automatically, it allows for faster code to be written with less effort.

Figure 5.1 visualises the samples drawn from 3 different specifications of the Neal's Funnel model: the non-efficient specification that uses centred parameterisation (Figure 5.1a), the efficient specification as presented in Stan's manual (Figure 5.1b), and the translated from SlicStan version (Figure 5.1c). We see that the translated program, even though syntactically different from the program given in the manual, is also able to sample well from the neck of the funnel. Performance-wise, we do not observe a significant difference between the two implementations. We draw 10 million samples in 1000 runs from each of the original and the translated models. The mean time for a run of the original model is 0.146 ± 0.005 seconds, and the mean time for a run of the translated model is 0.150 ± 0.003 seconds.[5]

[5]The error bar denotes 2 standard deviations from the mean.



(a) Non-efficient form. (b) Efficient form (manual). (c) Translated from SlicStan.

Figure 5.1: 24,000 samples obtained from 6 runs of Stan's sampler (default settings) for different Stan programs defining the Neal's Funnel model. The code for both (a) and (b) are taken from Stan's Reference Manual, while (c) is the Stan program resulting from translation of the model written in SlicStan.

"Neal's Funnel" translated to Stan

```
transformed data {
  real m;
  real mp;
  m = 0;
  mp = 0;
}
parameters {
  real x_raw;
  real x_rawp;
}
model{
  x_raw ~ normal(0, 1);
  xr_rawp ~ normal(0, 1);
}
generated quantities {
  real s;
  real sp;
  real x;
  real y;
  s = 3;
  y = (s * x_raw + m);
  sp = exp(y * 0.5);
  x = (sp * x_rawp + mp);
}
```

SlicStan gives the programmer the opportunity to write shorter, more concise code, and to reuse parts of it, which in turn can lead to making fewer mistakes. However, there are several advantages of the original Stan code, that the translated from SlicStan Stan code does not have.

The first advantage is that the original version is considerably shorter than the translated one. This is due to the lack of the additional variables `m`, `mp`, `s`, and `sp`, which, in the translated code, are a consequence of the *elaboration* step described in §§ 4.3.2. When using SlicStan, the produced Stan program acts as an intermediate representation of the probabilistic program, meaning that the reduced readability of the translation is not necessarily problematic. However, the presence of those additional variables does not only make the program less readable, but can, in some cases, lead to a slower inference algorithm. This problem can be tackled by introducing standard optimising compilers techniques, such as variable and common subexpression elimination.

Furthermore, we see that the translated code is not ideally optimised — the variable `s` could have been placed in the **transformed data** block. This happens, because type inference in SlicStan is done before elaboration of the program, and thus the type levels of functions' arguments and local variables are deduced on the function level. The call `my_normal(0, exp(y/2))` forces the second argument of `my_normal`, `s`, to be of at least level MODEL, and thus level GENQUANT is deduced. This causes all variables produced in place of `s` as part of the elaboration process to be of level GENQUANT, including `s = 3`.

Finally, we notice the names of the new parameters in the translated code — `x_raw` and `x_rawp`. Those names are important, as they are part of the output of the sampling algorithm. Unlike Stan, with the user-defined-function version of Neal's funnel, in SlicStan the programmer does not have control on the names of the newly introduced parameters. One can argue that the user was not interested in those parameters in the first place (as they are solely used to reparameterise the model for more efficient inference), so it does not matter that their names are not descriptive. However, if the user wants to debug their model, the output from the original Stan model would be more useful than that of the translated one.

5.3.2 Eight Schools

Another example that demonstrates the usage of the non-centred reparameterisation function, together with some current limitations of SlicStan, is the "Eight Schools" example [Gelman et al., 2013, p. 119], in which eight schools study the effects of their SAT-V coaching program. The input data is $\mathbf{y}$ — the estimated effects of the coaching program for each of the 8 schools, and $\boldsymbol{\sigma}$ — the standard deviation of those observed effects. The task is to specify a model which accounts for errors, by considering the observed effects to be noisy estimates of the *true coaching effects* $\boldsymbol{\theta}$. Taking the noise to be Gaussian, and $\boldsymbol{\theta}$ to be a Gaussian variable with mean $\mu\mathbf{1}$, and deviation $\tau\mathbb{I}$, we have:

$$\mathbf{y} \sim \mathcal{N}(\boldsymbol{\theta}, \boldsymbol{\sigma}\mathbb{I})$$

$$\boldsymbol{\theta} \sim \mathcal{N}(\mu\mathbf{1}, \tau\mathbb{I})$$

Below, we show an implementation of this model adapted from Stan's GitHub page. On the right, we see that the Stan code uses non-centring reparameterisation to make sampling faster — $\boldsymbol{\theta}$ is declared as a transformed parameter, obtained from the standard normal variable $\boldsymbol{\eta}$.

"Eight Schools" in SlicStan

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

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

real mu;
real tau ~ gamma(1,1);
real[8] theta;

theta[1] = my_normal(mu, tau);
theta[2] = my_normal(mu, tau);
theta[3] = my_normal(mu, tau);
theta[4] = my_normal(mu, tau);
theta[5] = my_normal(mu, tau);
theta[6] = my_normal(mu, tau);
theta[7] = my_normal(mu, tau);
theta[8] = my_normal(mu, tau);

y ~ normal(theta, sigma);
```

"Eight Schools" in Stan

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

Example adapted from https://github.com/stan-dev/rstan/.

On the left, we see the example written in SlicStan, and once again making use of the non-centring reparameterisation function my_normal. However, in this case $\boldsymbol{\theta}$ is a vector of 8 independent normal variables, thus, given that my_normal works for univariate Gaussians, we need to reparameterise each element separately. As SlicStan, in its current version, does not support **for** loops, nor vectorisation of user-defined functions, the 8 function calls need to be written one by one. This is problematic, not only because it increases the amount of code and human effort required, but also because it cannot be implemented if the number of schools is input-dependent. In the future, we plan to extend SlicStan to support **for** loops and vectorisation of user-defined function, as further discussed in § 4.5, which will remove the above limitation.

5.4 Scaling

As a final point in evaluating SlicStan, we run two experiments to study how the performance of its compiler scales with the size of the input program. To do so, we generate synthetic SlicStan programs of different size, by randomly choosing statements, expressions, and distributions to build up the program. An example of a (short) synthetic program, together with its translation to Stan, is shown in Appendix B.

In the first experiment (§§ 5.4.1) we generate programs that do not contain any calls to user-defined functions. This means that in theory the elaboration step can be omitted, however the current implementation of SlicStan does not allow this, therefore the step is still executed (and performance results reported). In the second experiment (§§ 5.4.2), we look at synthetic programs, which can have calls to the non-centring reparameterisation function described earlier — my_normal. In each experiment we generate 600 programs in total — 40 programs for each of 15 program lengths chosen. The shortest program is 20 lines of SlicStan code, while the longest is 7000 lines. Each group of 40 consists of 4 subgroups of 10 programs with each subgroup having a different "number of variable declarations" to "number of statements" ratio (from 40% statements, to 75% statements).

5.4.1 Experiment 1: No Calls to User-Defined Functions

Figure 5.2 shows the time (in milliseconds) taken for programs of different sizes, that do not contain calls to user-defined functions, to be typechecked, elaborated and transformed to Stan code. We see that for programs of less than 1000 lines, translation of SlicStan to Stan is very fast — each of the three core steps takes less than 1 second. For the typechecking and elaboration time of larger programs, we observe a linear relationship on the log-log scale, which means a polynomial time with respect to the number of SlicStan lines.[6] We estimate the order of the polynomial to be approximately 2.27 and 2.35 for typechecking and elaboration respectively. Expressions and statements of elaborated SlicStan programs are identical to those of Stan programs, therefore the last step of the translation (transformation) involves only splitting statements and declarations into their appropriate Stan program blocks. We see evidence for this from the figure — translation time increases very little with the size of the input.

As this experiment does not involve programs containing calls to user-defined functions, the size of the resulting Stan program is approximately equal to that of the source SlicStan program. The dashed vertical line at 3610 lines of SlicStan code, which corresponds to roughly 3610 lines of Stan code, marks the size of the largest Stan source file we found on GitHub. We see that despite its observed polynomial behaviour, SlicStan's compiler can handle programs, whose translation is of the same size as the longest Stan program we found, in under 20 seconds.

[6] This, of course, is an average case empirical result for the set of synthetic programs we generate, and does not represent a worst-case scenario.

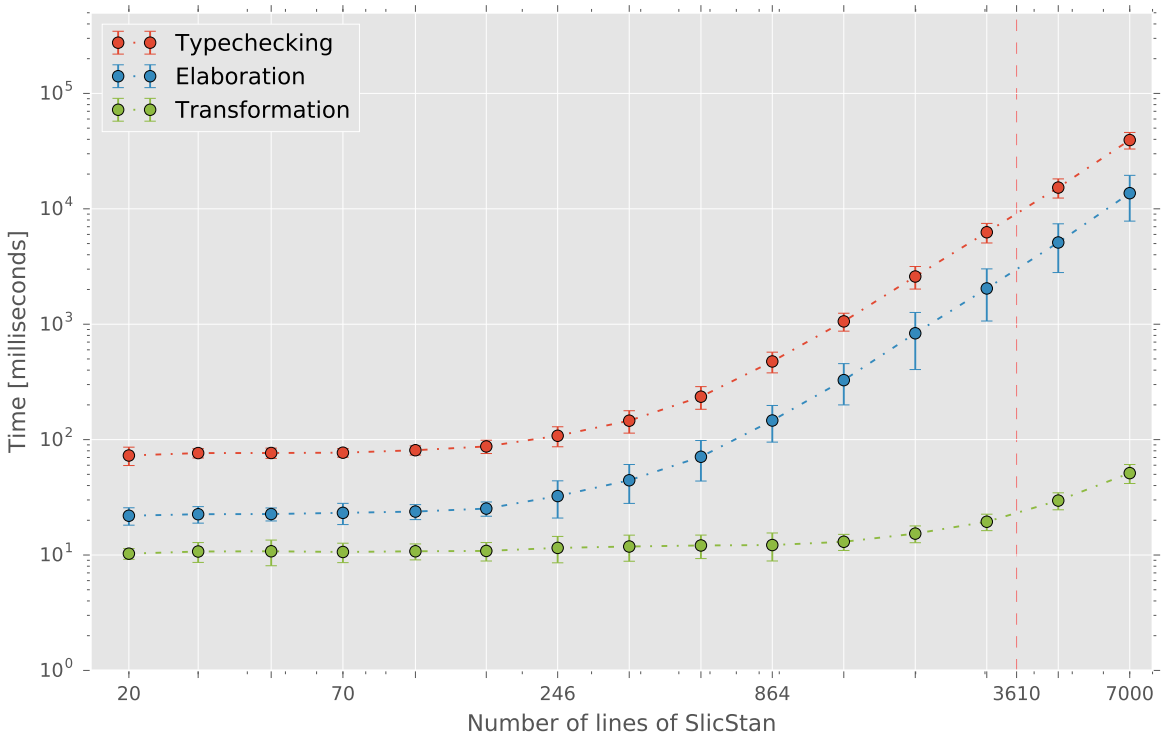


Figure 5.2: Mean time taken for SlicStan programs, without calls to user-defined function, to be typechecked, elaborated and transformed to Stan. The vertical bars show 2 standard deviations from the mean on each side. Both axes are in the log domain.

5.4.2 Experiment 2: Calls to a User-Defined Function

Figure 5.3 shows the time (in milliseconds) that the SlicStancompiler took to translate programs that contain calls to user-defined functions. For each generated program, approximately a third of the assignments contained a call to my_normal on the left-hand side. We observe a similar performance to that of Experiment 1, with one major difference — the elaboration time grows faster (polynomial order approximately 2.73).

On average, the ratio between the source SlicStan program and the resulting Stan program in this experiment is 0.58. This means that even though translating SlicStan programs of the same size took less time in the previous experiment, in Experiment 2 we obtain larger Stan programs. For example, the corresponding to the longest GitHub Stan length (3610) SlicStan length is approximately 2096 (marked with a dashed vertical line on the figure). SlicStan's compiler is fast in handling programs of this size — the figure shows a total translation time of approximately 7 seconds for a resulting Stan program of about 3610 lines.

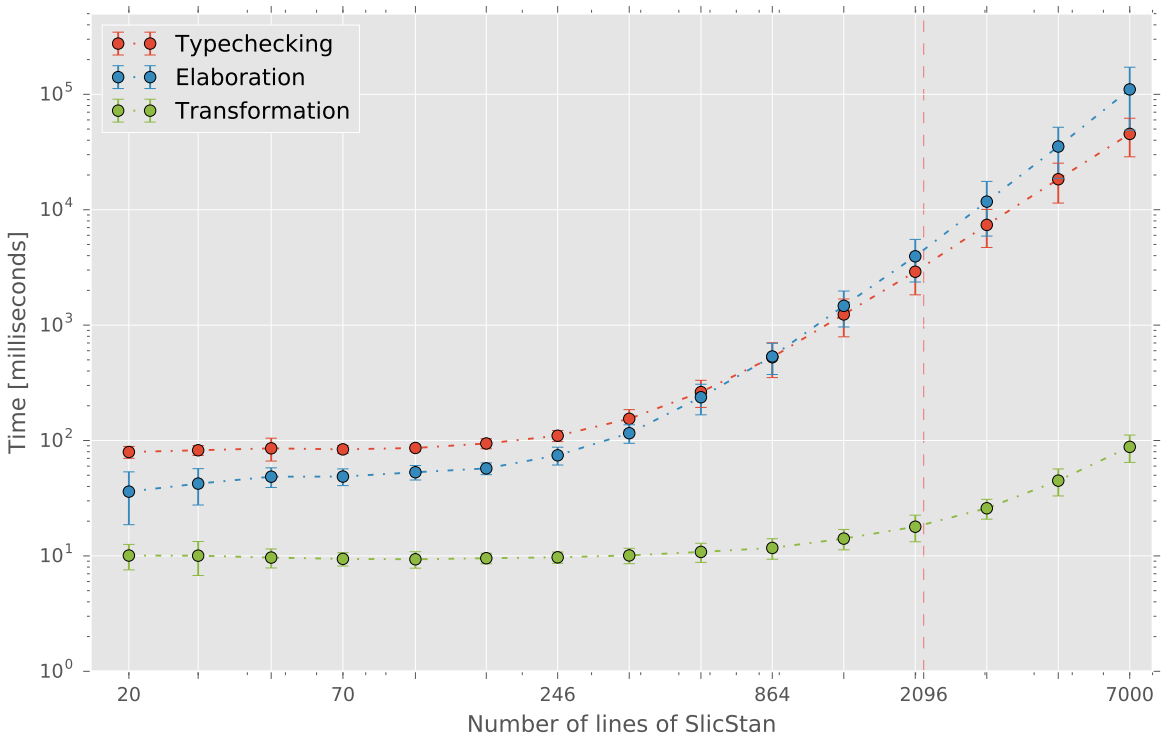


Figure 5.3: Mean time taken for SlicStan programs, containing calls to user-defined function, to be typechecked, elaborated and transformed to Stan. The vertical bars show 2 standard deviations from the mean on each side. Both axes are in the log domain.

Chapter 6

Conclusions

Similarly to ordinary programming languages, an ideal probabilistic programming language would be expressive enough to allow for a wide variety of models to be defined, yet abstract enough to free the programmer of the implementation details of the underlying inference algorithm. Probabilistic inference, however, 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.

The aim of this dissertation was to seek ways to apply existing programming language techniques to probabilistic programming, in order to reduce what needs to be done by the programmer, and allow for more abstract model specification. We achieved this by designing and implementing SlicStan — a probabilistic programming language, which compiles to Stan. SlicStan uses information flow analysis and type inference techniques to analyse the dependencies between program variables, and deduce their taxonomy classes. This allows the program to be compiled to Stan, which in turn uses those classes to generate an efficient Hamiltonian Monte Carlo sampling algorithm for the particular model. Furthermore, making the deduction of variable classes automatic, allows the language to be extended with more flexible user-defined functions. In particular, SlicStan's user-defined functions can introduce new model parameters, which, to our knowledge, is not currently possible in Stan.

We used examples adapted from Stan's Reference Manual and Stan's GitHub page to demonstrate the functionality of SlicStan. We showed that it allows Stan programs to be written in a shorter, more concise, and less viscous way. We also reported the time taken for large synthetic SlicStan programs to be translated to Stan. The results showed that the compiler is sufficiently fast to generate a Stan program longer than the longest Stan program we found on GitHub, within less than 30 seconds.

SlicStan successfully applies programming language techniques to allow Stan programs to be more concise, and easier to write and change. However, our work shows a more general point: programming language and static analysis techniques can be used to aid probabilistic inference. Such techniques have been applied to probabilistic programming before, but our work is novel in that it aids the Hamiltonian Monte Carlo algorithm, which is known to be efficient, but difficult to use and optimise. The Stan development

team's contribution to MCMC inference is crucial — they develop a system that uses the adaptive NUTS to make hand-tuning the parameters of the algorithm not necessary. We take one more step towards black-box Hamiltonian Monte Carlo inference — we automate the process of determining variables' taxonomy classes, and show that efficient automatic inference can be the result of the machine learning and programming languages communities joint efforts.

6.1 Future Work

Chapter 4 and Chapter 5 listed several limitations and possible extensions of SlicStan. For example, we discussed the possible implementation of direct access to the accumulated **target** log probability density function, as well as **for** loops, vectorisation of calls to user-defined functions, and language constructs for mixture models. Future work also includes proving properties of SlicStan's type system, and translation rules, as well as conducting a user study to learn more about how our language's usability compares to that of Stan. This section, on the other hand, discusses more general future directions of work.

Hierarchical Regression

Stan is widely used by statisticians and data scientists to build real-world hierarchical models. Language constructs such as Stan's **for** loops, are an intuitive way for the imperative programmer to specify multilevel regression models. This syntax, however, leads to long, hard-to-read programs. At his talk earlier this year, Gelman [2017] argues that in order to have large and complicated models that are applicable to real-world problems, we need more powerful syntax for probabilistic programming languages. He points at lmer [Bates et al., 2014] as an example of syntax that is one step closer to bringing statistical models the conciseness and simplicity they need, but argues that there should be an even more simple and intuitive solution.

A possible future direction for SlicStan is to explore ways in which regression models can be defined more concisely, and how those definitions can be translated to Stan, or other languages.

Automatic Reparameterisation

The concept of reparameterisation was previously outlined in §§ 2.3.2 and § 5.3. We saw the Neal's Funnel example, where strong non-linear correlations exist between different parameters of the probabilistic model. Sampling from such distributions using Hamiltonian Monte Carlo is challenging, because the distribution's curvature varies, and it is difficult to choose algorithm parameters (step size, path length, and mass matrix) that work well in all regions of the density.

The Neal's Funnel example is only one example of the difficult geometries that may arise in hierarchical probabilistic models, and the specific non-centring Gaussian reparameterisation is only one way to tackle this. Papaspiliopoulos et al. [2007] develop a more general

framework for the parameterisation of hierarchical models, and study what parameterisation — centring or non-centring — is more effective in different situations. They conclude that neither is uniformly effective, and that the two strategies can be combined so that they complement each other. For example, centring and non-centring parameterisation can be interweaved to boost MCMC, as suggested by Yu and Meng [2011].

SlicStan's flexible functions introduce an opportunity for an easier, and more concise implementation of parameterisation strategies, but this is not all they have to offer. Being able to express such strategies with functions provides a way to potentially automate the process. The compilation of a SlicStan program could be extended to apply appropriate semantic-preserving transformations that will lead to a more efficient parameterisation of the probabilistic model.

Program-Specific Choice of an Inference Algorithm

There exist numerous probabilistic programming languages and systems. Some of them, like Anglican [Wood et al., 2014], focus on the ability to express arbitrary probability distributions, no matter how complex there are. The price they pay to achieve this is using more general-purpose inference algorithms, that often are many times slower than optimised, special-purpose ones (on the problems that the latter can handle). Other languages, like Stan and Infer.NET [Minka et al., 2014], focus on efficient inference, but in most cases this means reducing the range of probabilistic models they provide support for, or the accuracy and guarantees of inference. Stan, for example, requires the target joint probability density of the model to be differentiable with respect to its parameters, and cannot handle discrete latent variables. Moreover, this means that Stan cannot handle variable-sized models (the number of parameters needs to be fixed, or be input-depended, but cannot be random). This makes it difficult, or even impossible, to express a wide range of probabilistic models in Stan.

This variability in the capabilities of different inference algorithms inspires another, more general direction of future work. If we can automatically optimise a probabilistic program for HMC inference, is it possible to analyse it and decide if HMC is the best algorithm to use for that particular program? Can we use static analysis to decide on what the most fitting inference approach is, and compile to a different back-end language, or straight to an optimised algorithm, depending on the type of model this program describes?

SlicStan, in a way, uses Stan as an intermediate representation of the probabilistic program. It would be interesting to extend the range of models SlicStan supports, and decide on a different intermediate language to use based on the particular model. For example, Anglican and its general purpose Sequential Monte Carlo inference algorithm, may be a way to allow for variable-sized models to be defined, while Infer.NET and Expectation Propagation may be an efficient way to work with models containing discrete variables.

Bibliography

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 147–160. ACM, 1999.

D. Bates, M. Maechler, B. Bolker, S. Walker, et al. lme4: Linear mixed-effects models using Eigen and S4. *R package version*, 1(7):1–23, 2014.

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

M. Betancourt and M. Girolami. Hamiltonian Monte Carlo for hierarchical models. *Current trends in Bayesian methodology with applications*, 79:30, 2015.

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

J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 77–96. Springer, 2011.

J. Borgström, A. D. Gordon, L. Ouyang, C. Russo, A. Ścibior, and M. Szymczak. Fabular: Regression formulas as probabilistic programming. In *ACM SIGPLAN Notices*, volume 51, pages 271–283. ACM, 2016a.

J. Borgström, U. D. Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *ICFP 2016 Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 33–46. ACM, 2016b.

S. Chasins and P. M. Phothilimthana. Data-driven synthesis of full probabilistic programs. In *International Conference on Computer Aided Verification*, pages 279–304. Springer, 2017.

G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 92–102. ACM, 2013.

T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

A. Gelman. Developer talk. https://www.youtube.com/watch?v=DJ0c7Bm5Djk&t=22m57s, *StanCon*, 2017.

A. Gelman and J. Hill. *Data analysis using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, 2007. ISBN 9780521686891.

A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC Press, London, third edition., 2013.

A. Gelman, D. Lee, and J. Guo. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5): 530–543, 2015.

M. Girolami, B. Calderhead, and S. A. Chin. Riemannian manifold Hamiltonian Monte Carlo. *arXiv preprint arXiv:0907.1100*, 2009.

N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and D. Tarlow. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

A. D. Gordon, T. Graepel, N. Rolland, C. Russo, J. Borgstrom, and J. Guiver. Tabular: a schema-driven probabilistic programming language. In *ACM SIGPLAN Notices*, volume 49, pages 321–334. ACM, 2014a.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014b.

M. I. Gorinova, A. Sarkar, A. F. Blackwell, and D. Syme. A live, multiple-representation probabilistic programming environment for novices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 2533–2537. ACM, 2016.

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

C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. *arXiv preprint arXiv:1701.02547*, 2017.

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

C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In *ACM SIGPLAN Notices*, volume 49, pages 133–144. ACM, 2014.

C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 186–195. IEEE, 1989.

D. Kozen. Semantics of probabilistic programs. *Journal of computer and system sciences*, 22(3):328–350, 1981.

A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei. Automatic differentiation variational inference. *arXiv preprint arXiv:1603.00788*, 2016.

D. Lunn, C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter. *The BUGS book: A practical introduction to Bayesian analysis*. CRC press, 2012.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4): 325–337, 2000.

D. J. MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

V. Mansinghka, R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves. BayesDB: A probabilistic programming system for querying the probable implications of data. *arXiv preprint arXiv:1512.05006*, 2015.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21 (6):1087–1092, 1953.

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

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

R. Neal. Probabilistic inference using MCMC methods. *Toronto: University of Toronto, 1993.–144 p*, 1993.

R. M. Neal. Slice sampling. *Annals of statistics*, pages 705–741, 2003.

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

A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI*, pages 2476–2482, 2014.

A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *ACM SIGPLAN Notices*, volume 50, pages 208–217. ACM, 2015.

O. Papaspiliopoulos, G. O. Roberts, and M. Sköld. A general framework for the parametrization of hierarchical models. *Statistical Science*, pages 59–73, 2007.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

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

G. Smith. Principles of secure information flow analysis. *Malware Detection*, pages 291–307, 2007.

Stan Development Team. Stan modeling language: Users guide and reference manual. http://mc-stan.org, 2017.

S. J. Taylor and B. Letham. Forecasting at scale, 2017. https://facebookincubator.github.io/prophet/.

D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

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

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

Y. Yu and X.-L. Meng. To center or not to center: That is not the questionan Ancillarity–Sufficiency Interweaving Strategy (ASIS) for boosting MCMC efficiency. *Journal of Computational and Graphical Statistics*, 20(3):531–570, 2011.

Appendix A

Multilevel Linear Regression

The Stan code below specifies a varying-intercept model of the (noisy) radon levels $\mathbf{y}$ of N houses in J different counties. For each house i, the predictor x_i has value 0 if the radon measurement was taken in the basement, and 1 if it was taken on the ground floor. We are interested in predicting a_j — the radon level of county j. We assume that y_i is a noisy measurement of the true radon level $\hat{y}_i$, which in a function of the floor level of the measurement x_i, and the radon level of the county $a_{j[i]}$:

$$y_i = \beta x_i + a_{j[i]}$$

In Stan, we express this using a **for** loop and array indexing to specify the correct a_j.

```
data {
  int N; // number of houses
  int J; // number of counties
  int county[N]; // county[i] is the county house i belongs to
  vector[N] x; // x[i] is the floor in which the measurement in house i is taken
  vector[N] y; // y[i] is the radon measurement in house i
}
parameters {
  vector[J] a; // a[i] is the radon level of county i
  real beta;
  real sigma_a;
  real sigma_y;
  real mu_a;
}
model {
  vector[N] y_hat;
  for (i in 1:N)
    y_hat[i] = beta * x[i] + a[county[i]];

  a ~ normal(mu_a, sigma_a);
  y ~ normal(y_hat, sigma_y);
}
```

This model is described by Gelman and Hill [2007]. The Stan code is adapted from https://github.com/stan-dev/example-models/.

Appendix B

Synthetic Example

This appendix gives a 30 line SlicStan synthetic example, and its translation to Stan. The example was generated in the following manner:

(1) Repeat 30 times:

   (a) Choose to generate a data declaration, a non-data declaration, or a statement with probability $\frac{1}{6}$, $\frac{1}{3}$ and $\frac{1}{2}$ respectively.

   (b) If we choose to generate a data declaration, generate **data real** x, where x is a fresh variable name.

   (c) If we choose to generate a non-data declaration, generate **real** x, where x is a fresh variable name.

   (d) If we choose to generate a statement and if there are less than 3 variables previously declared, generate *Skip*. If we choose to generate a statement and if there are at least 3 variables previously declared:

      i. Choose from assignment and sampling statement with probability $\frac{1}{3}$ and $\frac{2}{3}$ respectively.

      ii. If we choose an assignment, generate an assignment to a unitary function (sqrt, exp, or log), a binary function (+, −, ∗, /, or pow), or my_normal with equal probability.

      iii. If we choose a sampling statement, generate a sampling statement (choose one of the distributions normal, gamma, cauchy, beta, or uniform).

(2) The final SlicStan program is the definition of my_normal concatenated with the body generated in the previous steps.

A synthetic SlicStan program

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

real a;
data real b;
real c;
real d;
data real e;
a = my_normal(b, d);
real f;
a = (d - f);
d = (c - b);
d ~ beta(b, c);
real g;
real h;
real i;
data real j;
data real k;
d = my_normal(f, h);
f = (k + b);
i = log(d);
real l;
h ~ beta(d, d);
real m;
real n;
h = pow(a,m);
real o;
n = sqrt(e);
m ~ beta(g, n);
c = my_normal(a, b);
h = exp(j);
real p;
```

The SlicStan program translated to Stan

```
data {
  real b;
  real e;
  real j;
  real k;
}
transformed data {
  real f;
  real n;
  f = (k + b);
  n = sqrt(e);
}
parameters {
  real g;
  real mpp;
  real xr;
  real xrp;
  real xrpp;
}
transformed parameters {
  real a;
  real c;
  real d;
  real h;
  real m;
  real mp;
  real mppp;
  real v;
  real vp;
  real vpp;
  m = b;
  v = d;
  a = (v * xr + m);
  a = (d - f);
  d = (c - b);
  mppp = f;
  vpp = h;
  d = (vpp * xrpp + mppp);
  h = pow(a,mp);
  mp = a;
  vp = b;
  c = (vp * xrp + mp);
  h = exp(j);
}
model {
  xr ~ normal(0, 1);
  d ~ beta(b, c);
  xrpp ~ normal(0, 1);
  h ~ beta(d, d);
  mp ~ beta(g, n);
  xrp ~ normal(0, 1);
}
generated quantities {
  real i;
  real l;
  real o;
  real p;
  i = log(d);
}
```