SlicStan: A Blockless Stan-like Language

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton

1. What is SlicStan?

SlicStan[1] is a probabilistic programming language that compiles to Stan and has Stan-like syntax. There are two main ways in which the two languages differ:

1. SlicStan contains no program blocks, nor any annotations as to what block a variable belongs to (other than what the input data to the model is).
2. SlicStan supports more flexible user-defined functions, which can declare new model parameters.

For example, the following is a valid SlicStan program:

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

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

data int N;
data vector[N] y ~ normal(mu_y, sigma_y);

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

This is equivalent to the model given on page 101 of the Reference Manual (Stan Development Team 2017c). We use the SlicStan compiler to translate the program to Stan, which results in the same Stan model:

```
stansource <- tostan(file="models/simple.slic")
writeLines(stansource)
```

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

}
transformed data {
    real alpha;
    real beta;
    alpha = 0.1;
    beta = 0.1;

}
parameters {
    real mu_y;
    real tau_y;
```

[1] Pronounced “slick-Stan”. SlicStan stands for “Slightly Less Intensely Constrained Stan”.

```
}
transformed parameters {
    real sigma_y;
    sigma_y = inv(sqrt(tau_y));

}
model {
    mu_y ~ normal(mu_mu, sigma_mu);
    tau_y ~ gamma(alpha, beta);
    y ~ normal(mu_y, sigma_y);

}
generated quantities {
    real variance_y;
    variance_y = pow(sigma_y,2);

}
```

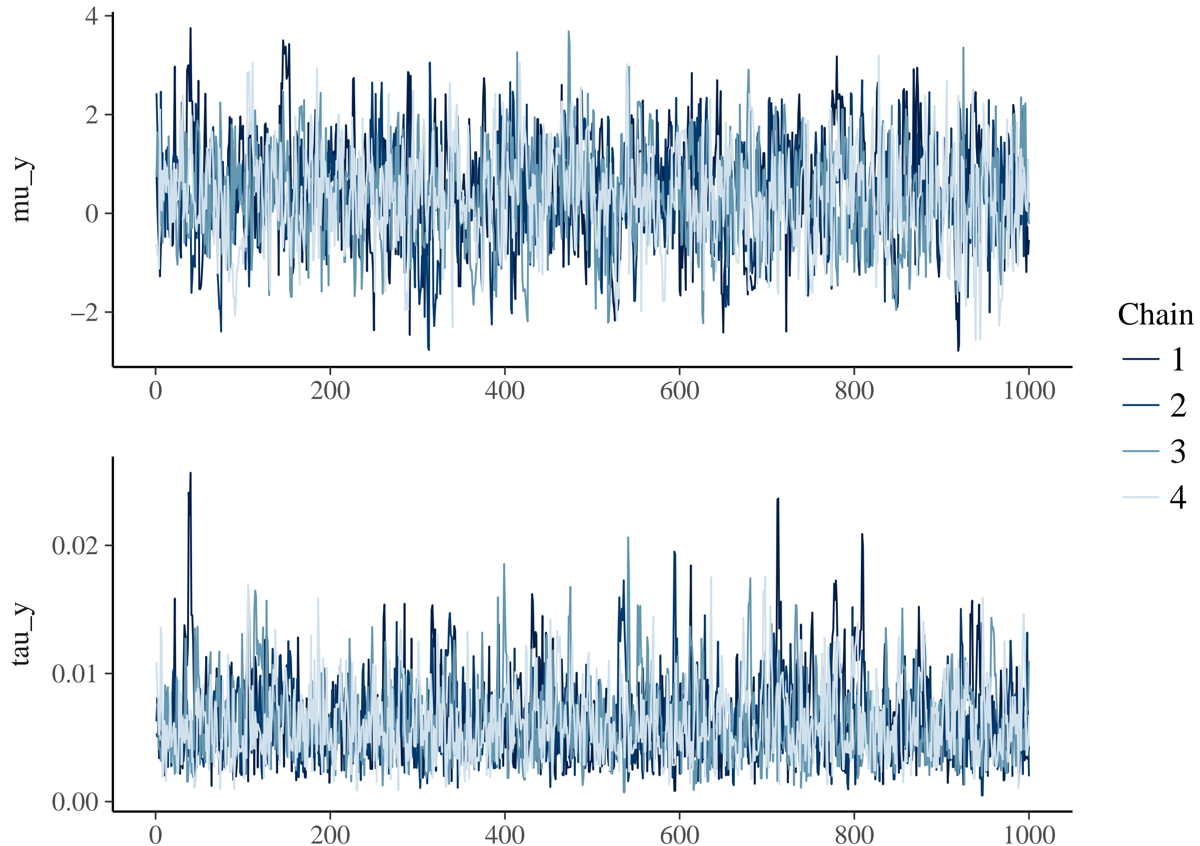
As the program is translated to Stan, it could readily be used with already available useful Stan tools, such as as bayesplot (2017a) and ShinyStan (2017b). For example, we can use RStan to compile the model above, and bayesplot to plot results or diagnostics:

```
library(rstan)
library(bayesplot)

data <- list(mu_mu = 0,
             sigma_mu = 1,
             N = 8,
             y = c(28,  8, -3,  7, -1,  1, 18, 12))

fit <- stan(model_code=stansource, data=data)

posterior <- as.array(fit)
mcmc_trace(posterior, pars = c("mu_y", "tau_y"),
           facet_args = list(ncol = 1, strip.position = "left"))
```



Motivation

There are several reasons why a *blockless* syntax, such as that of SlicStan, is desirable.

In Stan, 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 more or less efficient program. Declaring variables in the most suitable block could be challenging, especially for inexperienced users. Moreover, determining what that most suitable block is requires knowledge about the implementation of the underlying inference algorithm, as the block allocation is not always intuitive. For example, in the model from the previous subsection, the hyperparameters `alpha` and `beta` are declared as `transformed data`.

The presence of blocks makes it difficult to reason about a Stan program as a composition of other Stan programs, which may affect usability. A more compositional syntax can allow for better code reuse in the form of user-defined functions and libraries. It can also mean code that is easier to refactor. For example, extending an existing model will require adding code at one place of the program, rather than splitting it into chunks that belong to different blocks.

Finally, 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). Allowing functions to declare new parameters allows for better code reuse, and can lead to shorter, more readable code. For example, this would make it possible for libraries of common distributions to be implemented in pure Stan.

Currently, SlicStan supports only a small subset of Stan. For example, loops are not supported, neither are constraints on variables (such as `real<lower=0> sigma`). However, it is possible to extend the language with those and other constructs, as long as care is taken in terms of the static unrolling of functions (see section 2).

2. Compilation of SlicStan to Stan

SlicStan's compiler uses *information flow analysis* to deduce in what way the program should be *shredded* into program blocks. In short, each variable is associated with a *level type* — one of DATA, MODEL, or GENQUANT. Translation to Stan is then done in three steps:

- *Typechecking*: the SlicStan program is checked for type errors, and the level type of each variable is deduced.
- *Elaboration*: calls to user-defined functions are statically unrolled.
- *Transformation*: variable declarations and statements are shredded into different blocks, based on the level types deduced by typechecking, to produce the final Stan translation of the original program.

The rest of this section briefly discusses the idea behind information flow analysis and outlines the way each of the translation steps is implemented. A more elaborate explanation, including formal syntax and typing rules of SlicStan, is presented in the SlicStan MSc thesis (Gorinova 2017).

Information Flow Analysis and Level Types

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:

- PUBLIC — such variables hold low security, *public* data.
- SECRET — such variables hold high security, *secret* data.

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

- `P = P`
- `S = S`
- `S = P`

In other words, if we define an order $<$ on the levels, so that PUBLIC $<$ PRIVATE, secure information flow analysis is used to ensure that information flows only upwards with respect to that ordering. This is also known as the *noninterference property* — changes to confidential inputs lead to no changes in public outputs of a system (Goguen and Meseguer 1982).

In Stan, blocks must appear in a particular order, and each block can only access variables defined in itself, or a previous blocks. In other words, *information flows* from the first, `data` block, to the last, `generated quantities` block. We identify 3 *levels* associated with Stan's program blocks 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

Information flows from DATA through MODEL to GENQUANT, but not in the reverse order. In other words, changes to variables at level GENQUANT do not affect variables at level MODEL or DATA, and changes to variables at level MODEL do not affect variables at level DATA. Thus, we define the ordering:

DATA $<$ MODEL $<$ GENQUANT

We incorporate these 3 levels in SlicStan, by giving each variable a special *level type*. Then the question of whether or not a sequence of SlicStan statements can be split into Stan blocks that obey the ordering from above, is formalised as an *information flow analysis* problem.

Automatically Deducing Level Types

We implement SlicStan's type system according to Volpano, Irvine, and Smith (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.

Informally, the type system can be described by the following rules:

1. If `x` is declared *with* the keyword `data`, then `level(x)` = DATA.
2. If `x` is declared *without* the keyword `data`, and `x` is *not assigned to* anywhere in the program, then `level(x)` ≥ MODEL. In other words, if we never assign to a non-data variable, then this variable is either a parameter or an unused generated quantity.
3. If `x = foo(y)` for any function `foo`, then `level(x)` ≥ `level(y)`.
4. If `x ~ foo(y)` for any distribution `foo`, then `level(x)` ≤ MODEL and `level(y)` ≤ MODEL. In other words, if the target log density depends on a variable, then this variable cannot be a generated quantity.

These rules imply that *all* externally provided data must be declared using the keyword `data`. That is, all variables that would be declared in the `data` block in Stan, must be declared using the `data` keyword in SlicStan.

For example, consider the following SlicStan program:

```
real mu;
real sigma;
data real y ~ normal(mu, sigma);
real variance = sigma*sigma;
```

Following the intuition from above, we have:

- `y` is of level DATA (rule 1).
- `mu` and `sigma` are of at *least* level MODEL (rule 2), but also of at *most* level MODEL (rule 4).
- `variance` is of at *least* level MODEL (rule 3).

The variable `variance` can be either of level MODEL or of level GENQUANT (in contrast with `y`, `mu` and `sigma`, for which we found only one possible assignment of level types). Either level assignment will result in a program that satisfies the noninterference property, so we need to make a choice between the two levels based on something other than the ordering we discussed so far.

Looking at Chapter 6 of Stan's Reference Manual, 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 per chain, code associated with variables of level GENQUANT — once per sampled datapoint, and if the variables are of level MODEL — once per leapfrog (many times per sampled datapoint). This means that there is the following ordering of level types in terms of performance:

DATA < GENQUANT < MODEL

We implement *type inference* — the deduction of variables' level types — following the typing rules of SlicStan's type system and the observation on the difference in performance, so that:

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

For the simple example above, this means that we will chose `variance` to be of level GENQUANT rather than level MODEL, as GENQUANT < MODEL. Another way to think about this is: variable `variance` *can* be defined

as a generated quantity. Therefore, the target density does not depend on variable `variance` (follows from Rule 4 in the beginning of the section).

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.
- all statements that modify the target log density belong to the `model` block.

Returning to the example, we declare `y` in the `data` block, `mu` and `sigma` in the `parameters` block, and `variance` in the `generated quantities` block. And indeed, we can run the example through SlicStan's compiler to get this result:

```
code <- "
real mu;
real sigma;
data real y ~ normal(mu, sigma);
real variance = sigma*sigma;"

stansource <- tostan(model_code=code)
writeLines(stansource)
```

```
data {
    real y;

}
parameters {
    real mu;
    real sigma;

}
model {
    y ~ normal(mu, sigma);

}
generated quantities {
    real variance;
    variance = sigma * sigma;

}
```

Translating to Stan

Translation to Stan, in the cases when there are no user-defined functions, happens exactly as described in the end of the previous subsection — variable declarations and statements are shredded into different program blocks based on their level type.

In the cases when there are calls to user-defined functions, translation needs one more step. This is because functions in Stan cannot declare new parameters, and all parameters need to be declared in the `parameters`

block. Thus, in SlicStan, we need to statically unroll calls to user-defined functions.

Take for example a program, which contains a simple call to a `non_centred_normal` function that implements a non-centred reparameterisation of a single variable:

```
def non_centred_normal(real m, real s) {
    real x_raw ~ normal(0, 1);
    return s * x_raw + m;
}
real mu;
real sigma;
real y = non_centred_normal(mu, sigma);
```

This call is statically unrolled by introducing three new variables in the program — one for each argument `m` and `s`, and one for the local variable `x_raw`:

```
real mu;
real sigma;
real m = mu;
real s = sigma;
real x_raw ~ normal(0, 1);
real y = s * x_raw + m;
```

This *elaborated* program can then be translated to Stan, and we get:

```
code <- "
def non_centred_normal(real m, real s) {
    real x_raw ~ normal(0, 1);
    return s * x_raw + m;
}
real mu;
real sigma;
real y = non_centred_normal(mu, sigma);"

stansource <- tostan(model_code=code)
writeLines(stansource)
```

```
parameters {
    real x_raw;

}
model {
    x_raw ~ normal(0, 1);

}
generated quantities {
    real m;
    real mu;
    real s;
    real sigma;
    real y;
    m = mu;
    s = sigma;
    y = (s * x_raw + m);

}
```

The need to statically unroll calls to user-defined functions is one of the biggest current limitations of SlicStan,

as it is not possible to compile a program that contains loops with run-time bounds. SlicStan takes care to rename variables appropriately, so that variable names do not clash once functions are unrolled. However, the programmer has very little control on the names of local variables, meaning that both the resulting Stan program, and Stan's inference output can become less readable.

3. Examples

In this section, we demonstrate and discuss the functionality of SlicStan. We compare several Stan code examples with their equivalent written in SlicStan, and analyse the differences.

Example from Discourse

Another simple example showing SlicStan ability to deduce the Stan roles of variables automatically, is one adapted from a blogpost by Carpenter (2017), where he discusses the possibility of Stan changing in a similar blockless direction:

```
real alpha ~ normal(0, 10);
real beta ~ normal(0, 10);
real sigma_sq ~ inv_gamma(1, 1);
real tau = inv(sigma_sq);
data int N;
data vector[N] x;
vector[N] x_std = (x - mean(x)) / sd(x);
real sigma = sqrt(sigma_sq);
data vector[N] y ~ normal(alpha + beta * x_std, sigma);
```

This is a valid SlicStan program, and we can use the SlicStan compiler to translate to Stan:

```r
stansource <- tostan(file="models/bobs.slic")
writeLines(stansource)
```

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

}
transformed data {
    vector[N] x_std;
    x_std = ((x - mean(x)) / sd(x));

}
parameters {
    real alpha;
    real beta;
    real sigma_sq;

}
transformed parameters {
    real sigma;
    sigma = sqrt(sigma_sq);

}
```

```
model {
    alpha ~ normal(0, 10);
    beta ~ normal(0, 10);
    sigma_sq ~ inv_gamma(1, 1);
    y ~ normal((alpha + beta * x_std), sigma);

}
generated quantities {
    real tau;
    tau = inv(sigma_sq);

}
```

As also discussed in the original blogpost, the blockless syntax is preferable for several reasons:

- It is shorter. The Stan code can be written more compactly than the printed versions above, by combining some of the declarations with their respective definitions. But even then it is 25 lines, versus 9 lines of SlicStan code.
- It gives flexibility in terms of the order of declarations and statements. One can choose to write it so it follows a generative story, or exactly in the way the Stan program is written, but omitting the block names.
- It allows for related statements to be kept together. For example, `sigma` can be declared and defined just before it is used in `y ~ normal(alpha + beta * x_std, sigma)`.

Another important observation is that with SlicStan there is no need to understand where and how different statements are executed during sampling. The SlicStan code is translated to the same hand-optimised Stan code from the blogpost, 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` variable `x_std` in the `data` block, which would result in a program that does not compile. And even more subtly — they could have defined `x_std` and `tau` both in the `transformed parameters` block, in which case the program will compile to (semantically) the same model, however it will not be optimised.

However, not using blocks means that it is less obvious to the user what role each variable plays in the model. Once the constraints which blocks impose are removed, programs could become harder to debug.

In SlicStan, blocks are still present in a way — each variable has one of three level types (which are automatically deduced and hidden from the user) and can then be put into a particular Stan block based on whether or not the variable has been assigned to. This is important, because having the blocks present, even though implicitly, allows for them to be re-introduced to the user in a way that conveys their meaning less intrusively than having them as part of the syntax. For example, the deduced level types in a program could be available through an interactive editor that displays information when the mouse is hovering over a variable name, in the way that programming environments, such as Visual Studio, do. The signature of the target density, $p(data \mid parameters)$, could also be made available. But even without implementing a special interface, SlicStan compiles to Stan code, which can be debugged and modified separately.

Cockroaches

This next example aims to demonstrate how the removal of program blocks can lead to an easier to refactor code. The *Cockroaches* example is described by Gelman and Hill (2007), 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 $\boldsymbol{y}_{rep}$, and compare it to the actual data $\boldsymbol{y}$ to find that the variance of the simulated dataset is much lower than that of the real dataset. To account for this overdispersion, they introduce a overdispersion parameter $\boldsymbol{\lambda}$:

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

The code below shows the Cockroaches example in SlicStan before (without including the commented lines) and after (including the commented lines) adding the overdispersion parameter:

To implement the first Cockroach model, the one that does not account for overdispersion, in SlicStan, we write:

```
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(log_expo
                       + beta[1]
                       + beta[2] * roach1
                       + beta[3] * treatment
                       + beta[4] * senior
                       // + lambda
                       );
```

In Stan, that same model is:

```
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(log_expo
                  + beta[1]
                  + beta[2] * roach1
                  + beta[3] * treatment
                  + beta[4] * senior
                  // + lambda
                  );
}
```

In both cases, including the commented code into the model gives us the second Cockroach model, the one that includes the overdispersion parameter $\boldsymbol{\lambda}$.

Modifying the model to account for overdispersion requires adding a similar number of statements in both languages. However, in SlicStan those statements can be added anywhere in the program (as long as they appear before the statement modelling y, which depends on $\boldsymbol{\lambda}$). Stan, on the other hand, introduces an entirely new to this program block — `transformed parameters` — to calculate the deviation σ from the precision hyperparameter τ.

Neal's Funnel

With this next example, we demonstrate SlicStan's user-defined functions. The program below defines a non-centred parameterisation function, and calls it twice to specify Neal's funnel (see Neal (2003)):

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

real y = non_centred_normal(0, 3);
real x = non_centred_normal(0, exp(y/2));
```

This model is equivalent to the non-centred Stan model given in the Reference Manual (Stan Development Team 2017c):

```
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);
}
```

The SlicStan definition of the non-centred funnel is only longer than its centred version (`real y ~ normal(0,3); real x ~ normal(0, exp(y/2))`), due to the presence of the definition of the function. The SlicStan code is also very easy to read, as it hides away the declaration of the raw parameters introduced as part of the reparameterisation. In comparison, Stan requires defining the new parameters `x_raw` and `y_raw`, moving the declarations of `x` and `y` to the `transformed parameters` block, defining them in terms of the parameters, and changing the definition of the joint density accordingly.

As elaboration of calls to user-defined functions produce new variables, the SlicStan funnel model translates to a different program than the original Stan program given in the manual:

```
stansource <- tostan(file="models/neals.slic")
writeLines(stansource)
```

```
transformed data {
    real m;
    real mp;
    m = 0;
    mp = 0;

}
parameters {
    real x_raw;
    real x_rawp;

}
model {
    x_raw ~ normal(0, 1);
    x_rawp ~ normal(0, 1);

}
generated quantities {
    real s;
    real sp;
    real x;
    real y;
    s = 3;
    y = (s * x_raw + m);
    sp = exp((y / 2));
    x = (sp * x_rawp + mp);

}
```

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, 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 is 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.

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`. 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 level types in SlicStan are deduced before elaboration of the program, and thus the type levels of functions' arguments and local variables are deduced on the function level. The call `non_centred_normal(0, exp(y/2))` forces the second argument of `non_centred_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, in SlicStan's version of the non-centred funnel, 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.

4. Conclusion

We present SlicStan — a blockless Stan-like probabilistic programming language, which supports user-defined functions that can declare new model parameters. SlicStan successfully applies static analysis to allow Stan programs to be more concise, and easier to write and modify. This opens a range of possibilities for future work that uses programming language techniques to aid probabilistic programming and inference.

For example, a future version of SlicStan could support vectorisation of user-defined functions, which would allow for models to be specified even more compactly. The full ten-dimensional version of Neal's funnel, can then be simply written as:

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

real y = non_centred_normal(0, 3);
vector[9] x = non_centred_normal(0, exp(y/2));
```

Another possible direction of work is to extend the type system so that certain constraints on the range that parameters can have could be automatically deduced using type inference. For example, consider the following truncated Gaussian model (Section 12.2 of the Reference Manual):

```
real mu;
real sigma;

data int N;
data real U;
data vector[N] y ~ normal(mu, sigma) T[,U];
```

The data length, `N` must be non-negative, as it is used to define the size of a vector. The variable `sigma` must be positive, as it is used as the standard deviation in `normal(mu, sigma) T[,U]`. Finally, the datapoints `y` must be smaller than or equal to `U`, as we specified that they follow a truncated normal distribution.

Extending SlicStan's type system to support constraints, could allow us to automatically transform the SlicStan program above to the following Stan code:

```
data {
  int<lower=0> N;
  real U;
  real<upper=U> y[N];
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  y ~ normal(mu, sigma) T[,U];
}
```

Finally, semantic-preserving transformations could be applied, depending on the specific program, to compile to a better optimised and better behaved sampling algorithm. For example, loops could be collapsed to vectorised statements where possible, and common subexpressions could be aggregated automatically.

References

Carpenter, Bob. 2017. "Generalizing the Stan Language for Stan 3." http://discourse.mc-stan.org/t/generalizing-the-stan-language-for-stan-3/1599.

Gelman, Andrew., and Jennifer Hill. 2007. *Data analysis using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Goguen, Joseph A, and José Meseguer. 1982. "Security Policies and Security Models." In *Security and Privacy, 1982 Ieee Symposium on*, 11–11. IEEE.

Gorinova, Maria. 2017. "Probabilistic Programming with SlicStan." http://homepages.inf.ed.ac.uk/s1207807/files/slicstan.pdf.

Neal, Radford M. 2003. "Slice Sampling." *Annals of Statistics*. JSTOR, 705–41.

Sabelfeld, Andrei, and Andrew C Myers. 2003. "Language-Based Information-Flow Security." *IEEE Journal on Selected Areas in Communications* 21 (1). IEEE: 5–19.

Smith, Geoffrey. 2007. "Principles of Secure Information Flow Analysis." *Malware Detection*. Springer, 291–307.

Stan Development Team. 2017a. "Bayesplot: Plotting for Bayesian Models." http://mc-stan.org/bayesplot/.

———. 2017b. "Shinystan: Interactive Visual and Numerical Diagnostics and Posterior Analysis for Bayesian Models." http://mc-stan.org/shinystan/.

———. 2017c. "Stan Modeling Language: User's Guide and Reference Manual." http://mc-stan.org; Version 2.17.0.

Volpano, Dennis, Cynthia Irvine, and Geoffrey Smith. 1996. "A Sound Type System for Secure Flow Analysis." *Journal of Computer Security* 4 (2-3). IOS Press: 167–87.