SlicStan: Improving Probabilistic Programming using Information Flow Analysis

Extended Abstract

Maria I. Gorinova
University of Edinburgh
United Kingdom
m.gorinova@ed.ac.uk

Andrew D. Gordon
Microsoft Research Cambridge
United Kingdom
adg@microsoft.com

Charles Sutton
University of Edinburgh
United Kingdom
csutton@inf.ed.ac.uk

1 Introduction

Probabilistic programming languages provide a concise and abstract way to specify probabilistic models, while hiding away the 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.

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[1] — a probabilistic programming language that compiles to Stan and uses information flow analysis to allow for more abstract and flexible models. SlicStan is novel in two ways: (1) it allows variable declarations and statements to be automatically *shredded* into different components needed for efficient Hamiltonian Monte Carlo inference, and (2) it introduces user-defined functions that allow for new model parameters to be declared as local variables.

2 Stan and SlicStan

Stan [2] is an imperative probabilistic programming language, with syntax similar to that of BUGS [3, 4], that compiles to an efficient *Hamiltonian Monte Carlo* (HMC) inference algorithm [5]. The language is increasingly used for real-world scalable projects in statistics and data science (for example, Facebook's Prophet [7]), and it is a convenient way to share and reproduce results of statistical models.

Even though increasingly mature, and very powerful, Stan sacrifices some of its usability to make automatic inference possible. One language design choice, made in order to make compilation to an efficient HMC algorithm possible, is the presence of *program blocks*. Data needs to be defined in a particular block, parameters in another; blocks must appear in order; changing the block a variable is defined in could result in a semantically equivalent, but more or less efficient program. Stan allows a wide range of models to be defined in its modelling language, but requires of the programmer to understand specifics of the underlying inference algorithm in order to write efficient code. The block syntax makes reasoning about a program as a composition of other Stan programs difficult, which also may affect usability.

Our goal is to use static analysis techniques to allow for a more compositional and flexible syntax of Stan, while retaining Stan's efficiency and a syntax that is natural for the statistics community. We achieve this by designing SlicStan to have a similar syntax to that of a subset of Stan, but allowing the interleaving of statements that would belong to different program blocks if the program was written in Stan. This makes the language more compositional, and allows for related declarations and statements to be kept close to each other, similarly to the Python libraries Edward [8] and PyMC3 [6]. The language also supports more flexible user-defined functions, as it is no longer restricted by the necessity to group parameter declarations in a specific block.

Below, we show an example of a Stan program (left), and the same program written in SlicStan (right). In both cases, the data `y` is an array of real numbers coming from a normal distribution with some unknown mean `mu` and standard deviation `sigma`, where `sigma` is defined in terms of the precision `tau`. We also define the variance `v` in terms of `sigma`.

Stan

```
data{
 int N;
 real y[N];
}
parameters{
 real mu;
 real tau;
}
transformed parameters{
 real sigma = pow(tau,-0.5);
}
model{
 tau ~ gamma(0.1,0.1);
 mu ~ normal(0,1);
 y ~ normal(mu,sigma);
}
generated quantities{
 real v = pow(sigma,2);
}
```

SlicStan

```
data int N;
data real[N] y;
real tau ~ gamma(0.1,0.1);
real mu ~ normal(0,1);
real sigma = pow(tau,-0.5);
y ~ normal(mu, sigma);
real v = pow(sigma,2);
```

The two programs consist of the same set of statements. However, due to the absence of blocks, SlicStan is more compact and allows statements to be written in any order,

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

PPS'18, January 9, Los Angeles, CA, USA

as long as variables are not used before they are declared. Using information flow analysis and type inference, SlicStan's compiler can infer what block each variable belongs to, and shred the program to produce the Stan code on the left.

3 Information Flow in SlicStan

3.1 Blocks in Stan

A Stan program consists of 6 blocks,[2] all of which are optional with the exception of the `model` block. Each block has a different purpose, and can reference only variables declared in itself or previous blocks. Below is a summary of the order blocks must appear in, and their purpose:

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

We define each block to have one of three levels, DATA, MODEL, or GENQUANT, as summarised in Table 1. Knowing that each block can only access variables declared within itself or a previous block, we see that information flows from variables of level DATA, through those of level MODEL, to those of level GENQUANT, but never in the opposite direction.

Block	Execution[3]	Level
`data`	—	DATA
`transformed data`	per chain	DATA
`parameters`	—	MODEL
`transformed parameters`	per leapfrog	MODEL
`model`	per leapfrog	MODEL
`generated quantities`	per sample	GENQUANT

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

3.2 SlicStan's Type System

In SlicStan, we define a lattice $(\{\text{DATA}, \text{MODEL}, \text{GENQUANT}\}, \leq)$ of level types, where DATA $<$ MODEL $<$ GENQUANT. We use standard information flow typing rules [9] to ensure that in a well-typed SlicStan program, information flows only in the direction outlined above. The only rule that needs care is that of the *model statement*, `x ~ foo(a, b)`. In Stan, such statements have the same meaning as incrementing a special variable `target`, which contains the accumulated log probability density: `target += foo_lpdf(x | a, b)`. This variable can be accessed only from the `model` block, thus we can see the model statement as an assignment to a MODEL level variable:

$$(\text{MODEL})\frac{\Gamma \vdash E : \tau, \text{MODEL} \quad \Gamma \vdash D : \tau, \text{MODEL}}{\Gamma \vdash E \sim D : \text{MODEL}}$$

Finally, to allow for level types to be automatically inferred, we implement type inference by generating constraints on the levels that variables may assume according to the type system, and finding a satisfying assignment. In many cases, the block to place a variable is not fully determined by the information flow between variables. For example, moving `transformed data` definitions of a well-formed Stan program to its `transformed parameters` block does not change the meaning of the program. Thus, we refer to Table 1 once more, to see that code associated with different blocks is executed different number of times. This gives us a *performance ordering* on level types: DATA $<$ GENQUANT $<$ MODEL, meaning that, in terms of performance, it is preferable if a variable is of level DATA, and it should be of level MODEL only if needed.

3.3 Translation to Stan

Having resolved the type inference constraints and inferred level types of the variables in the SlicStan program, we can translate it to Stan. This is done in two steps.

1. *Elaboration*: calls to user-defined functions are statically unrolled. This is needed, as different local variables might need to be defined into different Stan blocks.
2. *Transformation*: variable declarations and statements of the elaborated SlicStan program are shredded into different program blocks depending on the level types of the variables involved, whether variables have been assigned to elsewhere in the program, and whether statements access the `target` variable.

Implemented this way, SlicStan supports more-flexible user defined functions compared to Stan. Unlike Stan, user-defined functions in SlicStan can declare new parameters, meaning that the code for a large set of desirable transformations (such as model reparameterisation) can be better reused.

4 Conclusions and Future Work

We designed SlicStan, formalised its type system and translation rules, and implemented a compiler that turns SlicStan to Stan. The language successfully makes use of 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, allowing for models to be specified even more compactly. Semantics-preserving transformations could be applied, depending on the specific program, to compile to a better optimised and better behaved sampling algorithm.

[2] Excluding the functions block, which we omit for simplicity.

[3] 'Chain', 'sample' and 'leapfrog' refer to different parts of the sampling algorithm. There are many leapfrogs per sample and many samples per chain.

References

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

[2] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

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

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

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

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

[7] Sean J Taylor and Benjamin Letham. 2017. Forecasting at Scale. (2017). https://facebookincubator.github.io/prophet/.

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

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