Getting more out of Stan: some ideas from the Haskell bindings

Thomas A Nielsen[1], Dominic Steinitz[1] and Henrik Nilsson[2]

1. Tweag I/O
2. School of Computer Science, University of Nottingham

Introduction

Probabilistic programming is one of the most promising developments in statistical computing in recent times. By combining programming languages theory with statistical modelling, it empowers modellers with a flexible framework for statistical computing in which a great variety of models can be expressed using composition of arbitrary probability distributions within flexible control flow to express dependencies and data structures known to, or hypothesised by, the modeller.

Probabilistic programming is distinguished from stochastic programming by the capability to condition the random variables on observed data; essentially to perform the Bayesian update and move from prior to posterior distributions. Sometimes the posterior suffices. For instance, a causal link may be predicated on a certain regression coefficient being non-zero. Nevertheless, probabilistic programming in its full power is not restricted only to calculating the posterior. Computations based on this posterior may be more directly relevant to the data analyst or to the decision maker and often require further probabilistic calculations. For instance:

- predicting outcomes based on new observations
- model criticism based on residuals or the posterior predictive distribution
- forecasting timeseries
- risk analysis
- decision-making
- resource allocation

Stan has emerged as the most practical and widely used probabilistic programming language for Bayesian computation. In its canonical form, Stan essentially only calculates the posterior. While some further calculations can be done within Stan in a generated quantities block, these are quite limited, necessitates repetition, and are coupled to the inference and thus cannot be calculated independently. Consequently, further analysis is usually carried out in a generic host programming languages like Python or R. Unfortunately, this often leads to repetition. For instance, to calculate residuals or to make predictions in a predictive model, one must write the model twice: once in the Stan modelling language, for inference, and once in the host language, for simulation and prediction. This has the disadvantage that the two models may become out of sync and some probability distributions may be parameterised in different ways. This is not much work for simple models, such as linear regression model; the reader may feel that the author doth protest too much. But for more complex models it may not at all be obvious how to transfer the posterior into simulation environment.

Here, we present the results of some experiments with creating bindings to Stan in Haskell (https://github.com/diffusionkinetics/open/tree/master/stanhs), a purely functional and statically typed programming language. Rather than present “yet another Stan binding” or even worse, try to persuade the reader to abandon their current programming language and learn Haskell, our aim here is to present some ideas that enable a richer set of probabilistic computations from a subset of Stan models. This obviates the need to also implement the model in the host language, thus addressing the above problem with existing bindings. Our ideas are general and could, in principle, be leveraged to improve existing interfaces to Stan. Nevertheless, we have chosen here to explore these ideas in Haskell due to its support for embedded languages, ease of re-factoring experimental code, and its emerging data science ecosystem.

The Haskell programming language

Haskell is a pure, statically typed, general purpose functional language with lazy evaluation. It is widely used in academia for teaching and research, and increasingly for commercial software development as Haskell is conducive to rapid development of reliable code. Purity means that functions have no side effects, except where explicitly reflected in the function type, and then only those effects that are permitted by the type in question. This enforced effect discipline, in combination with an expressive type system, make it a lot easier to understand and maintain code, especially when it grows large. Purity implies that equational reasoning is valid, making Haskell a particularly good fit for mathematical applications. Lazy evaluation means that computation is demand driven. This, to a large extent, frees the programmer from operational concerns, allowing them to focus on stating what to compute rather than how to do it, which is another reason for why Haskell is a good fit for mathematical applications.

One area where Haskell has proved particularly successful is as a host language for domain-specific languages. There are a number of reasons for this. One is Haskell's concise yet flexible syntax with extensive support for overloading of operators and other notation through type classes. Another is that all values in Haskell, including functions, are first-class; i.e., they can be bound to variables, passed to and returned from functions, and so on, greatly facilitating implementing appropriate domain-specific abstractions. Haskell is also particularly well suited for symbolic computations, such as those needed for compiler applications. In the setting of embedded domain-specific languages, this allows for a spectrum of implementation strategies, from interpretation to compilation, as well as programmatic construction of programs in the domain-specific language, often referred to as metaprogramming or metamodelling (Augustsson et al, 2008; Giorgidze and Nilsson, 2011; Svenningsson and Axelsson, 2013). Finally, thanks to its powerful type system, it is often possible to enforce domain-specific typing constraints (Thiemann, 2002). We will see some of these features being put to good use in the following.

For a data scientist or statistician, the Haskell language holds several attractions. Most importantly, Haskell now has

an ecosystem and a community for data science and numerical computing that is, if not as well developed as Python or R, then increasingly productive with many packages implementing different methods, in particular for a general purpose programming language that was not designed with numerical computing in mind. Haskell's lazy evaluation model can make it more difficult to reason about performance, this in practice rarely gets in the way of data science work and can be disabled in recent Haskell compilers. On the other hand, Haskell's type system makes makes it significantly easier to reason about and refactor code.

On a spectrum of data science needs, Haskell is particularly suited to productizing models that have been developed in languages or environments that may be more suited for exploratory data analysis. The inline-r (https://tweag.github.io/HaskellR/) project, for instance, gives Haskell programmers direct access to all of the R programming language, facilitating moving data science into a production environment. Haskell's type system makes it very simple to re-factor large code bases which may blur the boundary between data science and software engineering.

Bayesian modelling in Haskell and Stan

In our prototype interface, the Stan model itself is described in a data structure in the host language, here Haskell. This is unlike typical Stan interfaces, and has the disadvantage that the Stan file has slightly more syntactic noise and is less familiar. However, the Stan model description is now a value in a programming language that can be manipulated and calculated based on the circumstances, a form of metamodelling as discussed in the previous section. There are also other advantages that will become apparent later in this paper. Moreover, there are a number of ways in Haskell to reduce the syntactic noise by making the model look more like plain Stan code should that be desired in a more mature implementation; e.g. parsing the desired Stan syntax directly from a file or a "here document", i.e. an embedded multi-line string.

In our current implementation, the Stan model value looks like this:

```
linRegression :: [Stan]
linRegression = [
  Data [ lower 0 Int ::: "n"
       , lower 0 Int ::: "p"
       , Real ::: "y"!["n"]
       , Real ::: "x"!["n","p"]
       ],
  Parameters [
       Real ::: "beta"!["p"],
       Real ::: "sigma"
       ],
  Model [
          For "i" 1 "p" ["beta"!["i"] :~ normal (0.0,1.0)]
          ,"sigma" :~ gamma (1.0,1.0)
          ,For "i" 1 "n" [
            "y"!["i"] :~ normal (("x"!["i"] `dot` "beta"), "sigma")
          ]
        ]
  ]
```

In our implementation, Stan models are encoded as an abstract syntax tree, defined as an algebraic data type `Stan`. This makes it straightforward to generate or transform Stan models programmatically by writing functions working on abstract syntax trees using pattern matching. For instance, `normal` is a function taking a pair of numbers (mean, standard deviation) and returning the abstract syntax tree representing a draw from a normal probability distribution. Our encoding as a data type is similar to that employed in the more polished ScalaStan (https://github.com/cibotech/ScalaStan). Note that we do not yet support vectorised expressions, but this is not an intrinsic limitation of our approach. However, if two models are developed in the same Haskell module, then names will not clash even if they overlap as the Stan code will be treated separately.

In the above code, the syntax `::` indicates a type annotation. Here `linRegression` has the type `[Stan]`, a list of values of type `Stan`, an algebraic data type with constructors (variants) Data, Parameters, Model corresponding to sections of a Stan model.

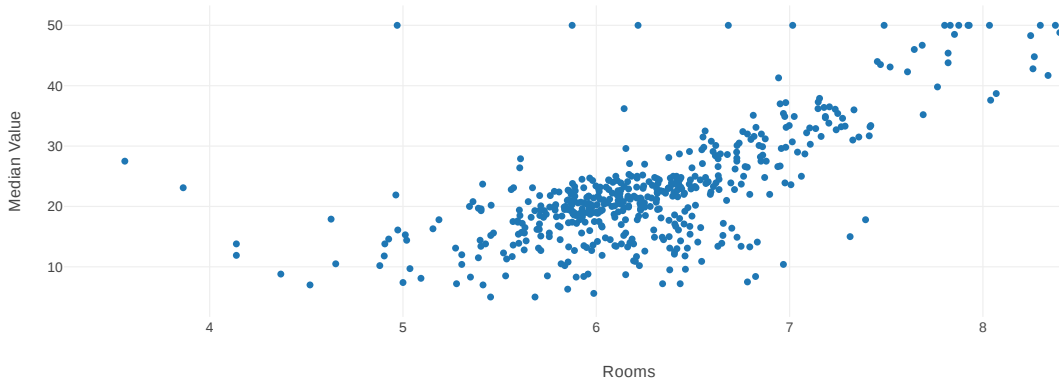
We can easily see the Stan code corresponding to the data structure (program) we just created by pretty-printing. This is just the Stan model file that we otherwise would use directly.

```
ppStans linRegression =>
data {
  int n;
  int p;
  real y[n];
  real x[n,p];
}
parameters {
  real beta[p];
  real sigma;
}
model {
  for (i in 1:p) {
    beta[i] ~ normal(0.0,1.0);
  };
  sigma ~ gamma(1.0,1.0);
  for (i in 1:n) {
    y[i] ~ normal(dot_product(x[i],beta),sigma);
  };
}
```

We then have to create the data structure holding the input to Stan. Here, as an example, we will use the Boston Housing dataset from the datasets (http://hackage.haskell.org/package/datasets) Haskell package. We load this into the `bh` variable which will hold a list of records describing Boston housing data (`bh :: [BostonHousing]` where BostonHousing is a record with fields including `rooms` and `medianValue`)

```
bh <- getDataset bostonHousing
```

Here, we plot the median value against the number of rooms in a residence:



To put this into the Stan data format, we create values of the `StanEnv` type using the custom infix operator `<~` . Haskell allows library programmers to define their own infix operators describing the model. Here, we have defined `v <~ d` to mean: "Create a Stan environment where the variable named `v` holds the data contained in the Haskell variable `d` ." `d` can be of any type for which we have defined how to turn values into Stan values (that is, implemented the `ToStanData` type class). We concatenate these elementary Stan environments using the append operator `<>` . `StanEnv` itself is represented as a map (known as a dictionary in some other languages) from strings, relating a variable name to its value. This environment is used both to contain data to be passed to Stan for inference and the variables generated in simulation (see below).

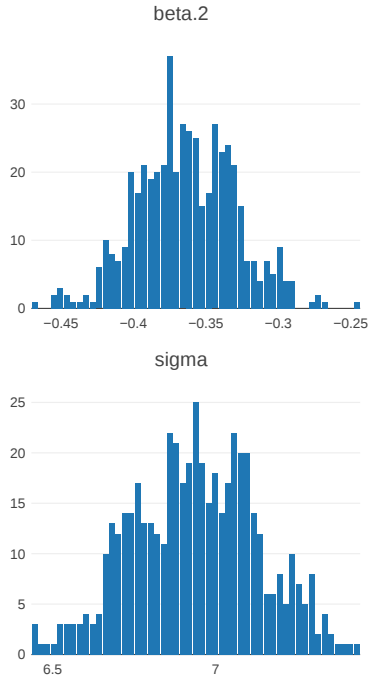
```
let getRow b = [rooms b, crimeRate b]

    sdata = "y" <~ map medianValue bh <>
            "x" <~ map getRow bh <>
            "n" <~ length bh <>
            "p" <~ (2 :: Int)
```

Finally, we run the Stan model using the `runStan` function taking as arguments the model, the data value and a configuration value that specifies whether we are sampling or optimising the posterior. The resulting posterior will be bound to the variable `res` . A

```
res <- runStan linRegression sdata sample {numIterations = 500}
```

At this point the components of the posterior can be plotted as usual:

beta.1



Simulating From a Stan Model

In order to obtain a richer probabilistic programming capability based on the Bayesian update in Stan, it suffices to add a function to simulate from a probabilistic model with fine control over which variables from the posterior are used in the simulation. By using no variables from the posterior at all (and therefore not using any data either), we are simulating from the prior (prior predictive distribution); by using all the variables from the posterior, we are simulating from the posterior (posterior predictive distribution). Moreover, by controlling the independent variables in the dataset, we can make predictions for new observations. In the case of timeseries modelling, by manipulating the starting value, we can continue a simulation from the endpoint of observed data (that is, forecast). Crucially, we propose that all of these functions are possible without writing the model twice as is usual: once in Stan, and once in the host language. Simulation operates on the same model description as that used for inference.

The type of our simulation function is:

```
runSimulate
  :: Int       -- Number of simulations we would like to perform
  -> [Stan]    -- The Stan model
  -> StanEnv   -- The simulation input environment
  -> [StanEnv] -- A list of independent simulation outputs
```

Here `runSimulate n m e` performs n independent simulations in the model m using environment e. If m contains values from a posterior (which has been generated with `runStan`), then each of the independent simulations uses a consistent set of samples from that posterior representing the state of the Markov chain at a given iteration. Therefore we retain information about posterior correlations in the simulations.

Internally, `runSimulate n` calls `runSimulateOnce n` times, simply collecting the results into a list. `runSimulateOnce` steps through the model definition's abstract syntax tree, potentially adding to the provided environment for every line. It will follow these rules:

- If the current line is a static assignment (corresponding to `<-` in the Stan syntax), calculate the right hand side and assign it to the variable given on the left-hand side in the environment
- If the current line is a draw from a probability distribution (corresponding to `~` in the Stan syntax), first check if the variable on the left-hand side is already present in the environment from the posterior.
- If the variable is present, select one of the iterations from the posterior and insert the posterior value at that

iteration into the environment. The iteration index selected is constant throughout one invocation of `runSimulateOnce`.

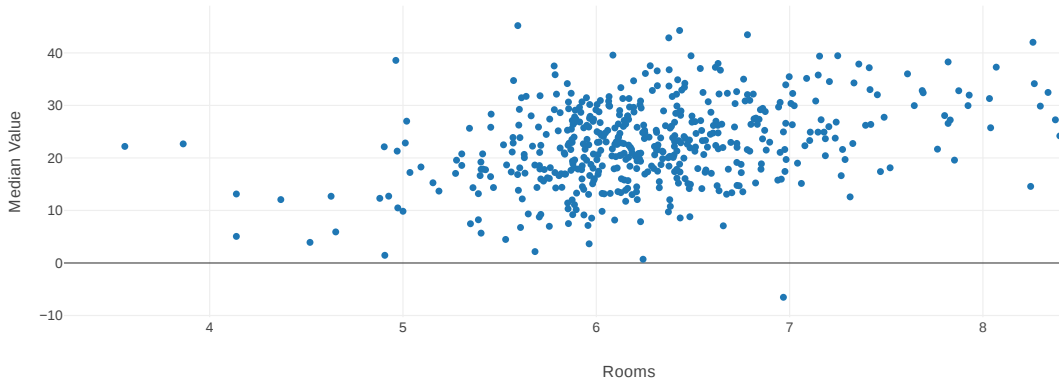
- If the variable is not present, fully evaluate the arguments to the probability distribution. Then use the random seed in the environment to simulate a draw from this probability distribution and a new random seed. Insert the drawn value and replace the random seed with the new seed.
- If the current line is a for loop, repeat the body for the required number of times with the loop index placed into the environment.

We quickly demonstrate the concept of simulation by simulating a single replica dataset and plotting it:

```
seed <- seedEnv <$> newPureMT     -- generate seed for random numbers
let resEnv = seed <> deleteVar "y" sdata <> mcmcToEnv res -- posterior and data with de
leted observation
    simEnv = runSimulateOnce linRegression resEnv  -- run the simulation
    postPredOnce = zip (unPairDoubles (lookupVar "x" simEnv))
                       (unDoubles (lookupVar  "y" simEnv))
```

In the first line, we create an environment holding a random seed to be used in a random number generators during the simulation. Then, we concatenate three environments to pass to the simulation: the random number seed, the data with the deleted observation variable `y` such that new variables with this name will be generated in the simulation, and finally the results of the posterior inference converted to an environment using the function `mcmcToEnv`. `runSimulateOnce` then runs the model in a simulation mode, taking as arguments the model representation and the environment holding the variables which are not to be simulated but are used as the basis for further simulation. In the last line, we use `zip` (which converts two lists to a list of pairs) to create a list of simulated `x` vectors, which are those in the data, and `y` which are the new simulated outcomes.

This can be plotted using the plotly library as:

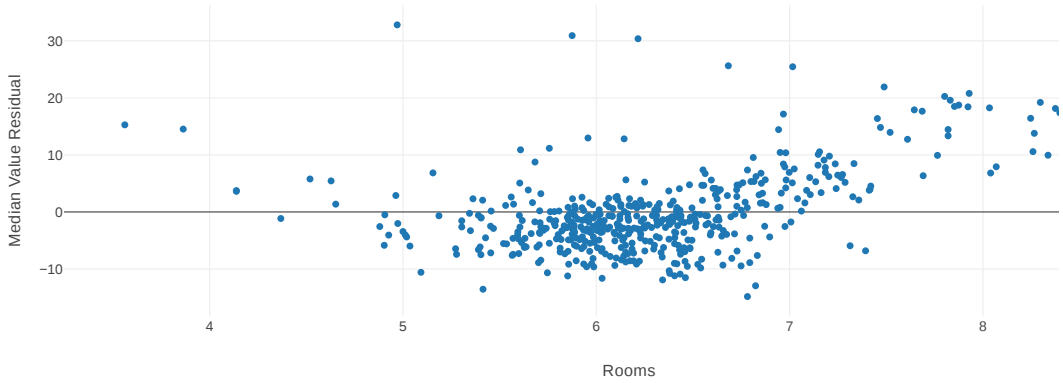


This simulation facility can be used for a common operation in model criticism: calculating residuals. In classical statistics residuals are calculated by averaging the posterior parameters (using a point estimate), making a single prediction and subtracting this from the observed outcome. From a Bayesian point of view, it almost never makes sense to average the parameters. Instead, the parameters, the predicted outcome and the residuals themselves are all probability distributions. In order to achieve something plottable, we average the predicted outcome over all the Markov chain Monte Carlo samples and subtract this average prediction from the observed outcome; but we could also use intervals or small point clouds for the full residual distribution. This has the advantage that in case of banana-shaped or multimodal posteriors (arising, for instance from lack of identifiability) the average prediction is more meaningful than the prediction with the average parameter value.

```
let -- simulate 100 datasets
    simEnvs = runSimulate 100 linRegression resEnv

    -- Average the 100 simulations into an average predicted outcome
    avgYs = avgVar simEnvs "y"

    -- for each average predicted outcome, subtract the observed outcome
    differences = zipWith (-) (unDoubles (lookupVar "y" sdata))
                              (unDoubles avgYs)


    -- zip the residual with the dependent variables to prepare for plot
    residuals = zip (unPairDoubles (lookupVar "x" sdata))
                    differences
```

Here we simulate 100 sets of predicted outcomes, that is 100 different outcome vectors `y`. For each simulated outcome vector index, we average the predictions to obtain a single average predicted outcome vector `avgYs`. From this average prediction, we subtract the observed outcome element wise in the list (`zipWith (-)`) and pair

these with the dependent variables to facilitate plotting.



Discussion

We have shown prototype bindings to Stan in the Haskell language. Unlike other such bindings, our model definition is a data type in the host language. This enables us to simulate from the model with fine control over the model parameters which may come from a posterior following inference.

Simulation with fine control over the provenance of the parameter distribution has several advantages. We have already shown how it can be used to define the residuals and how to simulate from the posterior predictive distribution. In many cases, such an instruction to simulate from the posterior predictive distribution will be incompletely specified. For instance, if we are dealing with a hierarchical regression model, should the posterior predictive distribution retain the group identities? One posterior predictive simulation would have entirely new groups, while another could have new individuals but drawn from groups that are specified from the data.

Further work

The Haskell interface to Stan we have shown here is an incomplete prototype. The model definition language can be polished to reduce the amount of syntactic noise. Ideally, we would also have the ability to parse models written in the standard Stan language. We can also make it much easier to move data into Stan, and to parse data from the posterior samples and from the simulation environment. This will be facilitated when the Haskell data science community converges on one of the several proposed implementations of data frames.

We currently only support a narrow subset of Stan models for post-posterior processing. Some features are simply not implemented in the work in progress we are presenting here, but we see no technical obstacles to doing so. This includes for instance vectorized expressions, or with a bit more work, ordinary differential equations. For models that have multiple assignments with `~` in separate lines to the same variable, corresponding to multiple observations, the semantics may be somewhat confusing as in the simulation we would effectively be skipping subsequent assignments as these would be treated as observed after they have been drawn on the first assignment. Therefore the simulated variables would only be influenced by the first draw, unless this is explicitly implemented in a different way.

Other models will be much more difficult to automatically convert to simulation code, for example models that rely on updating the log posterior using `+=` statements. Simulating posterior predictive data or making predictions from new independent variables from these models would probably require running a new Markov chain. Thus simulating from these models would be similar to simulating from the Ising model, which is typically done with MCMC.

Representing a Stan model in a data structure opens other possibilities. We plan to implement a function to combine two different models by specifying a list of the parameters that are common in the two models. Thus, two different manifestations of the same underlying constants can be investigated independently before the evidence is combined. This functionality would have to deal with name conflicts between different models which could arise when combining to independently developed models. This can be done by appending an identifier corresponding to the model provenance to any conflicting names.

About this document

The code for this document is hosted on GitHub (https://github.com/glutamate/stancon). It is written using the inliterate notebook format (https://github.com/diffusionkinetics/open/tree/master/inliterate) which allows a mixture of code, text writing, and the result of running code. Compared to Jupyter notebooks, it is less interactive (there is no caching) and it emphasises a cleaner, human- readable source format that can be checked into version control. Plots are generated with Plotly.js.

References

Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage DSL embedded in Haskell.

In ICFP '08: Proceeding of the 13th ACM SIG-PLAN international conference on Functional programming, pages 225-228, New York, NY, USA, 2008. ACM.

George Giorgidze and Henrik Nilsson. Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL. In Julio Mariño, editor, Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010), volume 6559 of Lecture Notes in Computer Science, pages 48–65, Springer-Verlag, 2011.

J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In Trends in Functional Programming (TFP) 2012, Revised Selected Papers, volume 7829 of Lecture Notes in Computer Science, pages 21–36, 2013.

Peter Thiemann, Programmable Type Systems for Domain Specific Languages, Electronic Notes in Theoretical Computer Science, Volume 76, 2002, Pages 233-251, ISSN 1571-0661.