ppl: A Type-level Bayesian Network

Kevin Li - UNI: zl2606

May 2, 2018

1 Abstract

The goal of this paper is to provide the motivation and a first-effort implementation of a type-level domain specific language (DSL) for describing Bayesian networks (BN) in the Haskell programming language [?]. The library is called `ppl`.

The scope of this paper is limited to "expert system Bayesian networks", in which the conditional probability distributions are user-defined, and posterior inference/sampling is our sole focus.

We will first begin by describing the motivation for the development of such a system. We will then introduce the syntax of the DSL and discuss the implementation of the system. Throughout, we will refer to toy and canonical examples in order to assist understanding. We will also discuss properties that arise from the syntax of the DSL, and finally evaluate the performance of this DSL by comparing it with other systems with respect to syntactic simplicity, speed, and memory requirements.

Future work can extend this system to parameter learning and potentially learning network structure. Other optimizations can be implemented, such as automatically reducing the model when conjugate priors are detected.

2 Introduction and Motivation

The whole point of this project primarily stems from the following idea; that graphical models should be graphical in nature, and that the implementation of graphical models should be easy (if we're drawing pictures on whiteboards then we should be able to "draw pictures" in code).

Hence, it makes sense to develop a language where the graphical nature of the model is conveyed through the syntax of the language.

Subsequent motivation come from the desire for correctness; the compiler should let us know when our model is obviously wrong, and they should let us know before run-time; existing interpreted programming languages do not support this.

Finally, we wish to embed this in an existing general purpose programming language in order to leverage existing features. Thus, it makes sense to implement this in the Haskell programming language due to its amenability to writing DSLs and powerful type-level programming.

3 System and Syntax

Throughout this paper, we will assume knowledge of Haskell syntax. Where there are code examples, the following imports will be required to run them.

```
import qualified Data.ByteString.Lazy.Char8 as BS

import GHC.TypeLits
import Data.OpenRecords
import Data.Proxy

import Model
import Model.PDF
import Model.Condition
import Model.Simulation
import Model.Types
import Model.Internal
import Model.Condition.Types
import Model.Simulation.Types

import System.Random
```

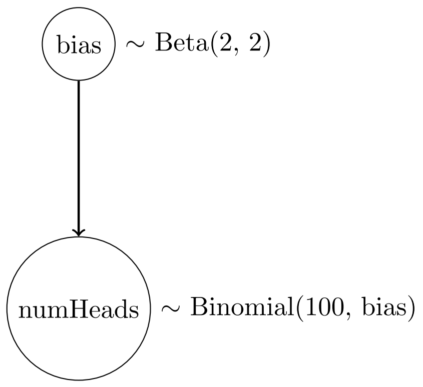
At a high-level, the desired workflow for performing inference is as follows:

1. Write down Bayesian network structure and variable types in a type using the DSL
2. Define the distributions and conditional probability distributions at the value level (i.e. the generative process)
3. Observe a value (i.e. condition on a node/several nodes taking on a value)
4. Sample from the posterior distribution of the rest of the network

We will describe at a high level the work that was done in order to achieve this workflow.
First, we define the syntax of the type level DSL. We do this in BNF form.

$$\phi, \psi ::= \texttt{n :=: t} \mid \texttt{n :=: t} \,\texttt{|->}\, \phi \mid \phi \,\texttt{:|:}\, \psi$$

where ϕ and ψ are Bayesian networks.
n :=: t defines a single node with variable name `n` and type `t`. These nodes are random variables with support over subsets of type `t`.
`n :=: t |->`ϕ defines a directed edge in the network, where the distribution of ϕ is a function of the variable `n` of type `t` (which must be its parent) in the network.
ϕ `:|:` ψ concatenates networks; in particular, ϕ cannot depend on ψ whereas ψ may depend on nodes in ϕ.



```
type MyNetwork =
    "bias" :=: Double
    :|: "bias" :=: Double |-> "numHeads" :=: Int
```

Figure 1: A Beta-Binomial coin flipping model and its corresponding type

As an example, see figure 1 for an example of a Bayesian network and its corresponding type. Note that in the graphical representation, we have explicitly written out the distribution of the prior and the conditional distribution, whereas the type only makes available dependence- and type-related information; this is intentional. Distribution information will be implemented at the value level.
At the value level, we implement the network in the following way.

```
model :: SimulationModel MyNetwork
model = beta 2 2 :|: (\p -> binomial 100 p)
```

Implementations of distributions over values of certain types, along with functions for sampling from these distributions and for obtaining their respective probability density functions is provided by an existing software library, *random-fu* in Haskell [1].
Note that in this context, the type `MyNetwork` has multiple potential value-level implementations; essentially, any 2-node network where the first generates a Double, and the second depends on the first and generates an Int is a valid value-level implementation of a network with this type. Any other implementation would not type-check and would cause the program to fail at compile-time.
After this model is written down, `ppl` allows the user to perform essentially two actions: simulation and posterior inference. We will discuss how `ppl` handles posterior inference internally, as that is the more interesting case.
First, we must perform validation and type-checking of the type itself. Symbol analysis is conducted to ensure that variables that are used (i.e. if a node depends on a previous node) have already been defined. Type-checking checks that the required dependencies have the correct type. We also check that variables are not re-defined (i.e. variables must have different names) and that acyclicality is obeyed (which follows by construction due to the way that the syntax is defined and how symbol analysis works).

Then, once the BN type is correct, the type family `SimulationModel` computes the required type of the generative model. See the source code for details. In particular, we require that each node of the network, given its parents, has both a `Distribution` and `PDF` instance (which are typeclasses in `random-fu`). This allows us to sample from each node and also evaluate the pdf/pmf at each node (given its parents).

Given this implementation, simulation is easy. All we have to do is sample from "top-to-bottom"; once we have accumulated samples from parent nodes, we pass them as arguments to later nodes, which instantiates the correct conditional distribution at child nodes (from which sampling becomes possible). We use the `CTRex` Haskell library [2] in order to have type-safe sample records; this allows us to correctly output samples and also prevents the user from conditioning on variables that are not defined in the network (which we will expand on later on).

As for posterior inference, in `ppl` we have opted to only implement the Metropolis-Hastings algorithm [3] due to time constraints and to facilitate generality. Internally, posterior inference is conducted the following way. First, the user specifies an observation within the network. This takes the form of conditioning on certain nodes taking on particular values. Then, the user must specify a jumping distribution for the entire network (including the nodes that have been conditioned upon for syntactic purposes). We have restricted, for the purposes of this software library the space of jumping distributions to ones where the marginal distribution of the target is conditionally independent of the other nodes in the target given the previous instantiation of the network. More precisely, let $\vec{x} \in \mathbb{R}^d$ be a vector containing the previous instantiations of a BN with d nodes. A jumping distribution in general is a conditional pdf

$$Q(\vec{x}^* \mid \vec{x})$$

where $\vec{x}^*$ is a new, proposed instantiation of the same BN. In `ppl`, we have that

$$Q(\vec{x}^* \mid \vec{x}) = \prod_{i=1}^{d} Q(x_i^* \mid \vec{x})$$

where x_i^* is the ith component of the vector $\vec{x}^*$.

We have chosen to include this restriction for simplicity of implementation and for sampling; additional work is required to allow for general sampling from arbitrary joint jumping distributions.

Finally, the user must specify an initial observation for Metropolis-Hastings. Using these, `ppl` proposes new samples from the jumping distribution, evaluates the acceptance probability, and then accepts or rejects the new sample according to the acceptance probability.

In Metropolis-Hastings, the acceptance probability requires the computation of the joint pdf of the BN. Bayesian networks can be factored into a product of conditional distributions (using conditional independence properties) [4], and the syntax of the network construction allows `ppl` to, given an instantiation of the network, evaluate the pdf at each node by instantiating its correct distribution given parent values, and then taking the product.

For example, consider the Beta-Binomial model in figure 1. Let $x =$ bias and $y =$ numHeads. Then the joint distribution can be written as

$$f(x, y) = g(x)h(y \mid x)$$

where $g(x)$ is the Beta(2, 2) density function, and $h(y \mid x)$ is the Binomial(100, p) density function where $p = x$. Given an instantiation of this network (i.e. realizations of x and y), we can compute the joint probability by simplying applying this function to the realizations.

For numerical purposes, we instead evaluate the sum of the log-pdfs.

4 Examples

4.1 Beta-Binomial

In this section we will implement some examples in `ppl` and observe the output.

The application we will focus on is the following: suppose that we have observed 25 heads out of 100 coin flips. What is the posterior distribution of the bias?

First, we will implement the Beta-Binomial model in figure 1. The code for the value-level implementation of the network has already been written, so all we have to do is write down the jumping distribution and a starting point for the Metropolis-Hastings (MH) algorithm.

We will define a jumping distribution where the bias is perturbed by adding a Normal(0, 0.01) random variable (where 0.01 is the variance) to the original bias. We will also need to define a jumping distribution for the `numHeads` node in order to obey syntax rules; of course, this will not be used by the library since we have conditioned on the `numHeads` variable being 25. Note that in `ppl`, the jumping distribution is called the proposal distribution.

We will also define a starting point for the MH algorithm; note that we will have to fully instantiate the network in this case; values must not be left undefined. In order to better showcase MH, we will start off with the proposal that the bias is 0.8.

First we need to include some boilerplate to make the type magic work.

```
-- | Necessary for type-level programming
p = Proxy :: Proxy Coins
-- | Necessary for MH algorithm
pConds = Proxy :: Proxy '["numHeads"]
-- |  Labels for the sample record.
numHeads = Label :: Label "numHeads"
bias = Label :: Label "bias"
```

And now we will implement the jumping distribution and the initial observation. Again, knowledge of Haskell syntax along with familiarity with the `CTRex` library will be assumed. We will also have to write down the data structure that will contain the condition that `numHeads = 25`.

```
-- | The "observation"
conds :: Sample Coins
conds = numHeads :<- 25 .| initSample p

-- | Jumping distribution
prop :: ProposalDist Coins
prop =
  (\prev -> normal (prev .! bias) 0.01)
  :|: (\prev -> binomial 100 0.5) -- will be unused

-- | Initial observation for Metropolis-Hastings
start :: Sample Coins
start = bias :<- 0.5 .| numHeads :<- 25 .| initSample p
```

Now, we can run the MH algorithm after providing the type proxies, condition, jumping distribution, and starting point.

```
main :: IO ()
main = do
  let g = mkStdGen 123 -- seed for reproducibility
  let samples = csim 10000 p pConds model prop conds g [start]
  putStrLn $ BS.unpack $ toCsvBS p samples
```

Running the program, we obtain the samples from the function `csim` and then print them to stdout by converting the samples from the MH algorithm into CSV format.

We can then analyze these samples to empirically check the posterior distribution. See figure 2 for a histogram of the posterior generated values of `bias`.

The data we obtain from the model yields the following statistics:

```
Sample Mean = 0.26508155076436174
Sample Var. = 0.002447678064961008
```

and so a 95% confidence interval for the true posterior mean is

$$(0.168, 0.362)$$

We also know from Bayesian statistics that a Beta(α, β) random variable is a conjugate prior for the p parameter for a Binomial(N, p) random variable, where the posterior distribution is given

$$\text{Beta}(\alpha + N_1, \beta + N_0)$$

where $N_1 + N_0 = N$, and N_1 is the number of ”successes”.

In our context, the posterior distribution is Beta(27, 77), and the posterior mean is $\frac{27}{104} = 0.2596$, which quite close to our simulation sample mean.

4.2 Bayesian Linear Regression

We will construct a toy Bayesian simple univariate linear regression model with only three observed data-points and with known conjugate priors in order to verify the performance of `ppl`.

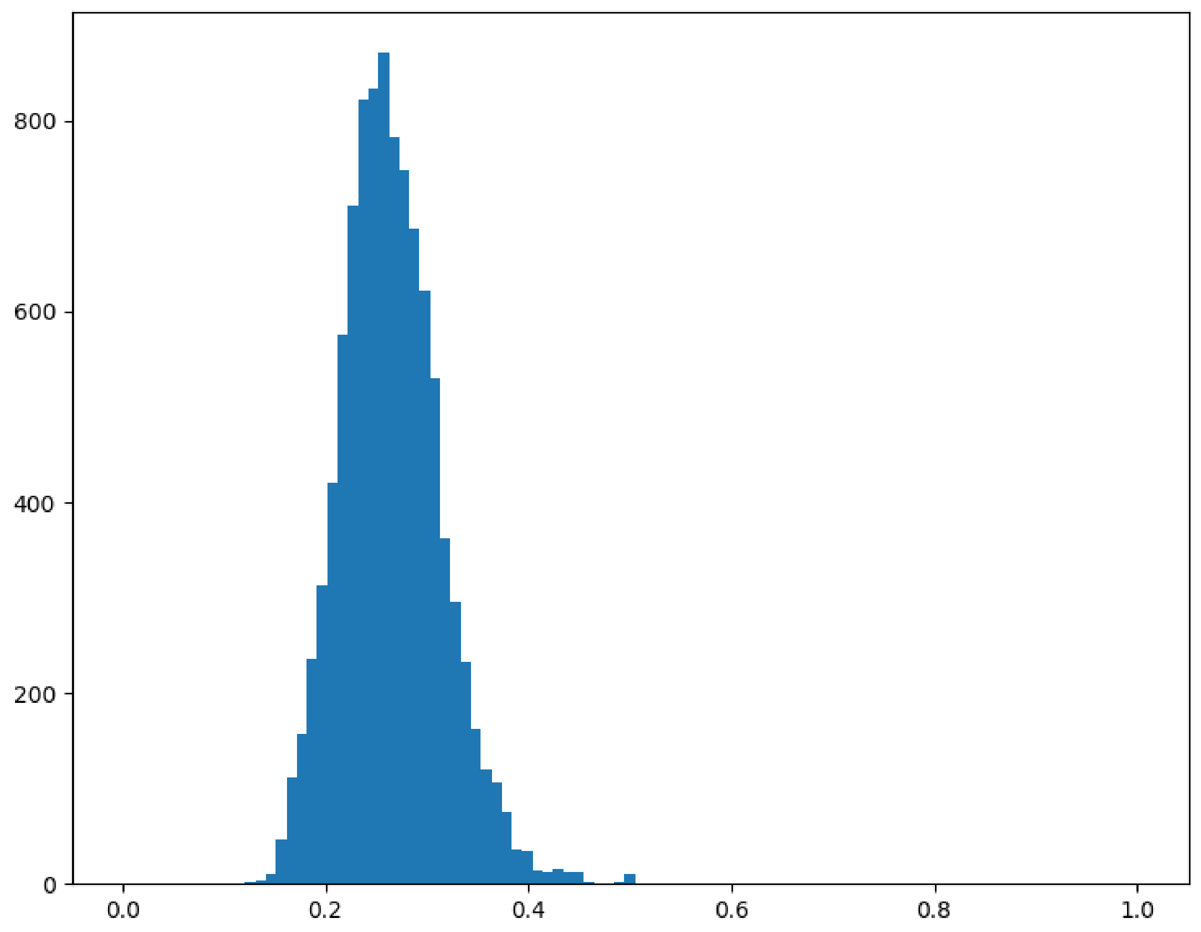


Figure 2: Histogram of 10,000 samples from the poterior of `bias` conditioned on the number of heads being 25 out of 100 coin flips

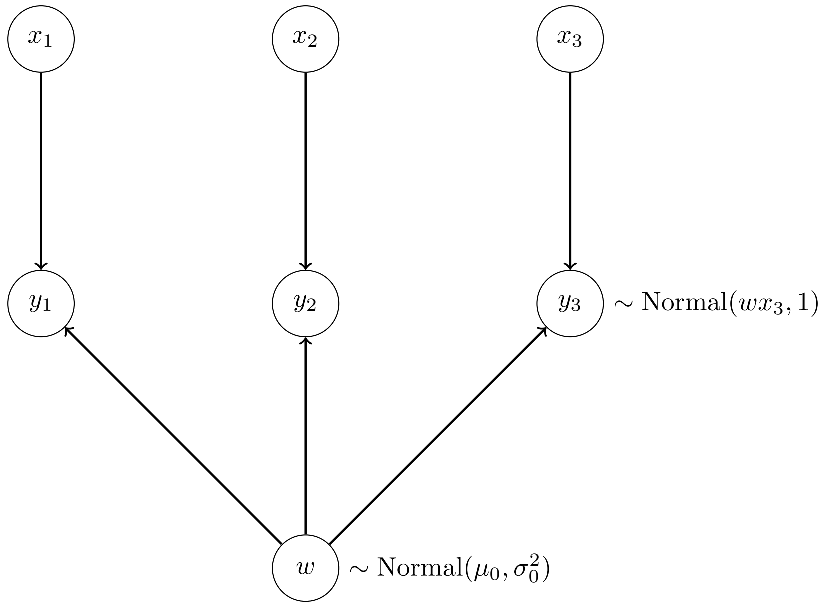


Figure 3: Bayesian simple linear regression as a graphical model

We will follow the exposition on Bayesian linear regression in Murphy [4] page 232. Since it is easy to estimate the bias term, we will assume that the y variables are centered.

We will assume that $x_1, x_2, x_3 \in \mathbb{R}$, $y_1, y_2, y_3 \in \mathbb{R}$, and $w \in \mathbb{R}$. The model is $y = wx + \epsilon$, and furthermore we will assume that the variance of ϵ is known, and that $\epsilon \sim$ Normal$(0, 1)$. Finally, we impose a Gaussian prior on w; specifically, that $w \sim$ Normal(μ_0, σ_0^2).

This model can be seen graphically in figure 3.

Note that in general, Bayesian linear regression does not have an analytical closed form for the posterior distribution of the parameter w. However, in this case, there exists an analytical solution for the posterior distribution of w. From Murphy, we have that the posterior distribution is Normal(μ_n, σ_n^2) where: [4]

$$\mu_n = \frac{\sigma_n^2 \mu_0}{\sigma_0^2} + \sigma_n^2 \sum_{i=1}^{3} x_i y_i$$

$$\sigma_n^2 = (\frac{1}{\sigma_0^2} + \sum_{i=1}^{3} x_i^2)^{-1}$$

With this in mind, we will implement the model for three datapoints. The toy data we have is

$$(0, 0), (2, 4), (4, 8)$$

which perfectly fits the line $y = 2x$.

First, we will write down the type of the BN and the value-level implementation of the generative model. Note that we place a Normal(0, 4) prior on the parameter w; no particular reason for this.

```
type LinearRegression =
  "w" :=: Double
  :|: "x1" :=: Double
  :|: "x2" :=: Double
  :|: "x3" :=: Double
  :|: "w" :=: Double |-> "x1" :=: Double |-> "y1" :=: Double
  :|: "w" :=: Double |-> "x2" :=: Double |-> "y2" :=: Double
  :|: "w" :=: Double |-> "x3" :=: Double |-> "y3" :=: Double

linRegModel :: SimulationModel LinearRegression
linRegModel =
  normal 0 4
  :|: constant 0
  :|: constant 2
  :|: constant 4
  :|: (\w x -> normal (w * x) 1)
  :|: (\w x -> normal (w * x) 1)
  :|: (\w x -> normal (w * x) 1)
```

Then, we have some necessary book-keeping:

```
lrp = Proxy :: Proxy LinearRegression
lrObs = Proxy :: Proxy '["x1", "x2", "x3", "y1", "y2", "y3"] -- we must correctly
w  = Label :: Label "w"
x1  = Label :: Label "x1"
x2  = Label :: Label "x2"
x3  = Label :: Label "x3"
y1 = Label :: Label "y1"
y2 = Label :: Label "y2"
y3 = Label :: Label "y3"
```

We condition on the data:

```
lrCond :: Sample LinearRegression
lrCond = x1 :<- 0
  .| x2 :<- 2
  .| x3 :<- 4
  .| y1 :<- 0
  .| y2 :<- 4
  .| y3 :<- 8
  .| initSample lrp
```

And now we must write down the jumping distribution for the parts of the network that haven't been conditioned upon; it is only necessary that we write a non-trivial jumping distribution for w. The jumping distribution for w is defined as the previous one plus a Normal$(0, \frac{1}{2})$

```
-- | Jumping distribution. Note
-- that if we have conditioned on certain values,
-- then it doesn't matter what the jumping
-- distribution for that value is.
lrJump :: ProposalDist LinearRegression
lrJump = (\prev -> normal (prev .! w) (1 / 2))
  :|: (\_ -> constant 0)
  :|: (\_ -> constant 0)
  :|: (\_ -> constant 0)
  :|: (\_ -> constant 0)
  :|: (\_ -> constant 0)
  :|: (\_ -> constant 0)
```

And finally we write down the starting point; note that here it also doesn't matter where the nodes that are not w start, since we have conditioned on all of them already.

```
-- | Reuse the condition record, just include
-- the 'w' term.
lrStart :: Sample LinearRegression
lrStart = w :<- 0 .| lrCond
```

We run the MH algorithm for 10,000 samples and plot the histogram in figure 4.

```
main = do
  let g' = mkStdGen 456
  let ws = csim 10000 lrp lrObs linRegModel lrJump lrCond g' [lrStart]
  putStrLn $ BS.unpack $ toCsvBS lrp ws
```

Recall the expressions for the posterior distribution of w. In our context, given our selections of the prior on w, we have for the posterior mean μ_n and posterior variance σ_n^2:

$$\sigma_n^2 = \left[\frac{1}{4} + 4 + 16\right]^{-1} = \frac{4}{81}$$

$$\mu_n = \frac{\frac{4}{81} \cdot 0}{4} + \frac{4}{81}(2 \cdot 4 + 4 \cdot 8) = 1.9753$$

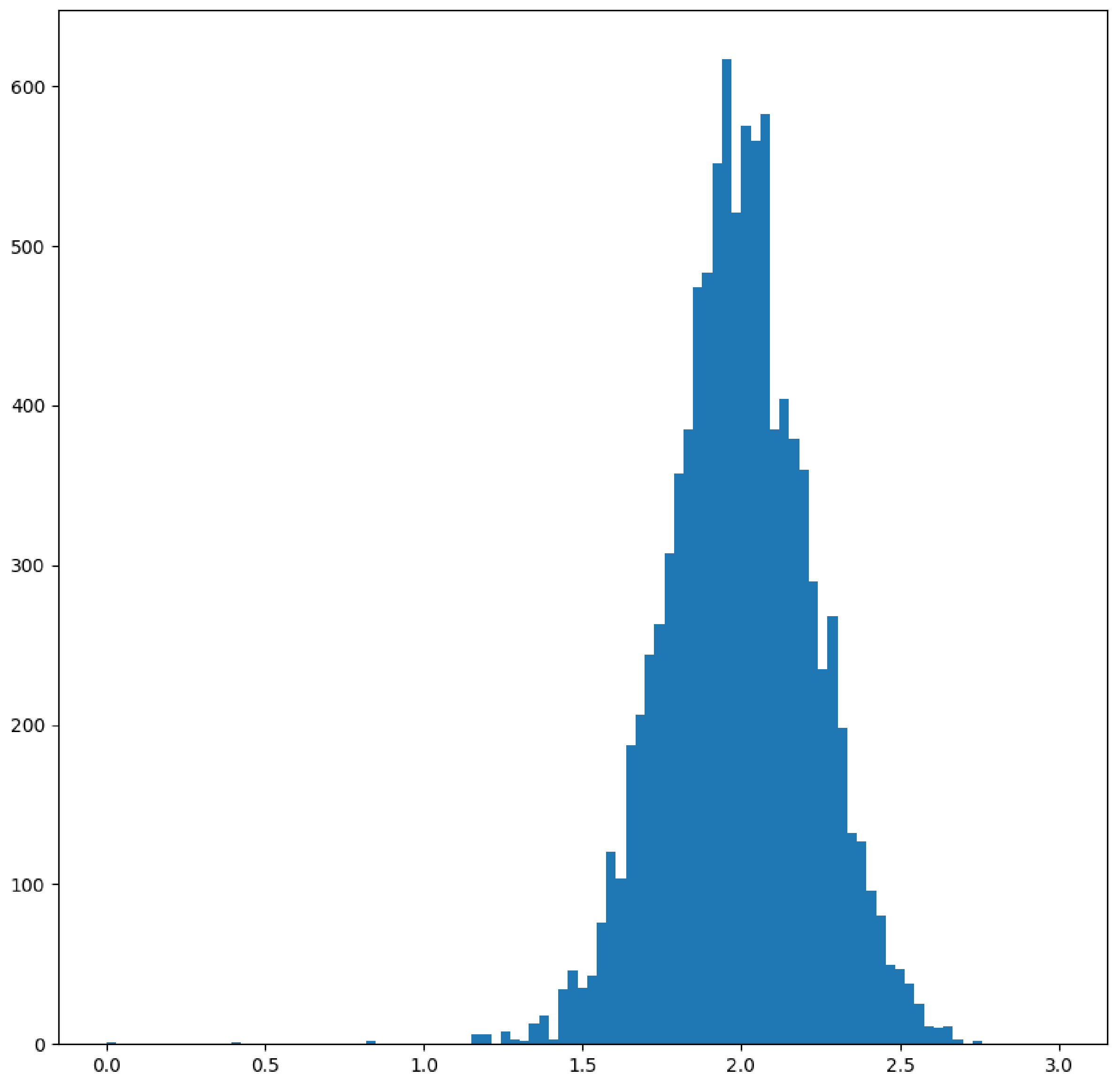


Figure 4: Histogram of 10,000 samples from the posterior distribution of the parameter w in Bayesian linear regression

In our simulation, we have a sample mean of 1.9912, and a sample variance of 0.0543, and a 95% confidence interval of (1.5345, 2.4479).

5 Conclusion and Extensions

We have shown that `ppl` can be used in several canonical scenarios. Type-safety allows the compiler to assist in identifying errors in model construction. Although we haven't explicitly identified errors in this paper, the reader can imagine situations in which a model is accidentally defined as being cyclic, or a node depends on an integer generating node when it really should be a real. Finally, the embedding of `ppl` in the Haskell programming language allows users to really take advantage of the general purpose programming language; currently in the library, we have already implemented a CSV exporting function for posterior samples. Other applications can very easily be built on top of `ppl`.

Although inference through the MH algorithm is indeed general, it is far from being a one-size-fits-all tool; furthermore, posterior inference isn't the only thing that one can do with Bayesian networks. There are many other tasks to take on in order to build this into a viable probabilistic programming library. Extensions include syntax for plate notation[4], allowing for generalization to n samples for applications such as Bayesian linear regression.

We need to also incorporate flat/improper priors, and include automatic optimization for conjugate priors (i.e. if we have an analytical solution then we shouldn't have to resort to numerical sampling).

Finally, work should be done on parameter learning and exact inference; there are many cases such as medical diagnosis where the nodes take on only discrete values, in which exact inference is tractable and fast; clearly Metropolis-Hastings is overkill in these scenarios. Exact inference algorithms should be implemented in future works, and parameter learning algorithms should also be implemented.

References

[1] James Cook. *random-fu:* Random number generation. https://hackage.haskell.org/package/random-fu. 2016.

[2] Atze van der Ploeg. *CTRex:* Open records using closed type families. https://hackage.haskell.org/package/CTRex. 2014.

[3] Gelman, Carlin, Stern, et al. *Bayesian Data Analysis, Third Edition.* 3rd edition. 2016.

[4] Kevin P. Murphy. *Machine Learning, A Probabilistic Perspective.* 2012, Massachusetts Institute of Technology.

[5] Davie, Antony (1992). *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press. ISBN 0-521-25830-8.