Declarative Modelling for Bayesian Inference by Shallow Embedding

[Work in Progress]

Henrik Nilsson
School of Computer Science
University of Nottingham
Nottingham, UK
nhn@cs.nott.ac.uk

Thomas A. Nielsen
OpenBrain Ltd
London, UK
tomn@openbrain.org

ABSTRACT

A common problem across science and engineering is that aspects of models have to be estimated from observed data. An instance of this familiar to control engineers is system identification. Bayesian inference is a principled way to estimate parameters: exploiting Bayes' theorem, an equational probabilistic model is "inverted", yielding a probability distribution for the unknown parameters given the observations. This paper presents Ebba, a declarative language for probabilistic modelling where models can be used both "forwards" for probabilistic computation and "backwards" for parameter estimation. The novel aspect of Ebba is its implementation: a shallow, arrows-based, embedding. This provides a clear semantical account and ensures that only models that support estimation can be expressed. As arrow-like notions have proved useful in modelling dynamical systems, this might also suggest an approach to an integrated language for modelling dynamical systems and parameter estimation.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (functional) programming; F.1.2 [**Computation by Abstract Devices**]: Probabilistic computation; I.6.2 [**Simulation and Modeling**]: Modeling languages

General Terms

Languages, Theory

Keywords

Modelling, Bayesian inference, shallow embedding, arrows

1. INTRODUCTION

Scientists and engineers are often faced with the situation that aspects of their mathematical models of physical phenomena and systems are uncertain as they are difficult or impossible to observe directly. These aspects then have to be inferred, or *estimated*, from what can be observed. The uncertainties often concern parameters of the model, but even the very model could be (largely) unknown, meaning that the best model from a set of candidates must be found, or even having to construct a model explaining the observations from scratch: "black-box" system identification [4].

Bayesian inference is a principled way to estimate parameters of probabilistic models given observed data [9]. Broadly, by applying Bayes theorem, the probabilistic model is "inverted", yielding a probability distribution for the parameters given the observations. A number of probabilistic modelling languages that support Bayesian inference exist. Examples include WinBUGS [5], Stan [10], and Baysig [6]. However, these languages tend to be stand-alone implementations. This paper investigates the possibility of implementing this type of language through *shallow embedding* [2, 11], using Haskell as the host language.

Embedded language implementations are of interest for a number of reasons. For the language designer and implementer, embedding offers a quick, low-effort approach to implementation and experimental language design [2]. Further, embedding in a general-purpose language automatically provides meta-programming facilities through the host language, yielding a much more powerful language at no extra implementation cost and obviating the need for end-users to learn an additional language for meta-programming. Integration of components written in a different languages within a single application is also greatly facilitated in the case where these language implementations are embedded in a common host language. Shallow embeddings have their specific pros and cons compared with deep embeddings [11]: a driving motivation here is that they offer a direct account of the semantics of the embedded language.

The starting point for the development in this paper is Baysig [6], a declarative, Haskell-like, functional language developed by OpenBrain Ltd that combines the model expressiveness of BUGS-like languages with general purpose computing. From the users' perspective, this is achieved by introducing a probability monad into a pure, functional language enabling probabilistic computations. A monadic value represents a random variable with some particular probability distribution, and the basic computation step is sampling of the distribution; i.e., drawing a value from it. In essence, this is no different from equipping an imperative language for probabilistic computation by adding a random

EOOLT 2014 Oct 10–10 2014, Berlin, Germany
ACM 978-1-4503-2953-8/14/10.
http://dx.doi.org/10.1145/2666202.2666208

number generator, except that the language remains pure and declarative as effectful computations (here, drawing a random value) and pure computations are carefully distinguished at the type level. Having added random number generation, generative, probabilistic models can be expressed in an intuitive and straightforward manner. Baysig allows such models to be run, generating observables according to the probability distribution specified by the model. Additionally, Baysig supports "inverting" models in the manner discussed above, computing probability distributions for the parameters of the model given actual observations. Basyig models are thus, in a limited sense, non-causal.

However, for the purpose of Bayesian parameter estimation, Baysig is arguably too powerful as it allows models to be expressed for which standard estimation techniques are not applicable. (Again, in essence, this situation is very similar to what would be the case in an imperative language equipped for probabilistic computation.) As will be explained later, the problem is that a monad is a too general notion of computation for this particular purpose, which raises the question if some other notion of computation might impose sufficient constraints to guarantee that standard techniques are applicable.

The standard estimation techniques work on models expressed as finite networks of random variables where the edges represent conditional dependences. In our setting, a random variable is represented by a value in the probability monad, and a conditional random variables by a function returning such a monadic value. Through the Kleisli construction, the latter is an *arrow* [3]. This suggests that something like arrows might be an appropriate abstraction for building "invertible" models, as arrows explicitly relates inputs to outputs, allowing us to keep track of the network structure, and as arrows can be constrained to ensure that the networks are finite.

In this paper, we introduce the language Ebba, short for Embedded Baysig, that takes an arrow representing a *conditional* probability distribution as the notion of computation rather than a monad representing a probability distribution. The two key contributions of this paper is that this approach enables:

- shallow embedding of a language for Bayesian inference
- combining probabilistic computation with a general-purpose language with a static guarantee that standard methods for Bayesian parameter estimation are applicable.

(Of course, similarly to solving equations numerically, there is no general guarantee that estimation actually will succeed.) While Ebba is still very much work in progress, and certainly nowhere near a language like Baysig in terms of capabilities or performance, our initial experience has been promising. Additionally, we hope that the readers not familiar with the technique of embedding for language implementation will find the paper interesting as an example what can be achieved through this technique and why this is useful.

Finally, we note that arrows and arrows-like notions have also proved useful for representing models of dynamical systems [7, 8]. While any connection is entirely superficial at this point, at least this allows us to think about the semantics of modelling languages for dynamical systems and Bayesian modelling languages using a common framework,

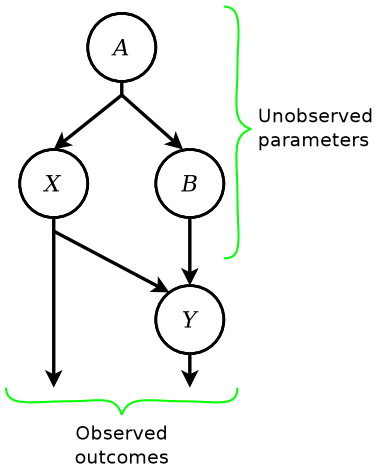


Figure 1: Probabilistic model

which in turn might suggest a way to integrate such languages, an open research question that has received relatively little attention [4]. However, Baysig does support stochastic differential equations, and there has been some work on stochastic differential-algebraic equations [1], which is an encouraging start.

2. IMPLEMENTATION

Bayes' theorem is often stated as

$$\mathrm{P}(X \mid Y) = \frac{\mathrm{P}(Y \mid X) \times \mathrm{P}(X)}{\mathrm{P}(Y)}$$

where

- $\mathrm{P}(X)$ is the *prior* probability
- $\mathrm{P}(Y \mid X)$ is the *likelihood* function
- $\mathrm{P}(X \mid Y)$ is the *posterior* probability
- $\mathrm{P}(Y)$ is the *evidence*.

Of central importance is that Bayes' theorem allows conditional probabilities to be turned around. This is exactly what we need to estimate unknown parameters given observations and a generative model, i.e., the distribution of the outcome given the parameters, as can be seen by reformulating Bayes' theorem as follows:

$$\mathrm{P}(\mathit{params} \mid \mathit{data}) = \frac{\mathrm{P}(\mathit{data} \mid \mathit{params}) \times \mathrm{P}(\mathit{params})}{\mathrm{P}(\mathit{data})}$$

Figure 1 shows a typical probabilistic model. The nodes are random variables, and edges between nodes represent conditional dependences. In this particular example, A, B are unobserved parameters, while X and Y are outcomes that can be observed directly. However, note that Y is conditional on X as well as on the unobservable parameter B. The (conditional) probability distribution for each variable (a probability density function (pdf) in case of a continuous random variable, a probability mass function (pmf) in case of a discrete random variable) is given. In figure 1, the

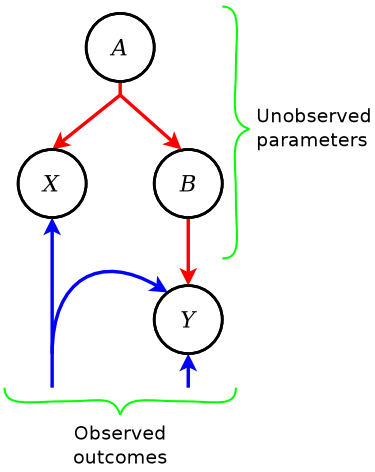


Figure 2: Estimator for probabilistic model

functions representing the distributions are

$$\begin{aligned} \mathrm{P}(A) &: T_A \to \mathbb{R} \\ \mathrm{P}(B \mid A) &: T_A \to T_B \to \mathbb{R} \\ \mathrm{P}(X \mid A) &: T_A \to T_X \to \mathbb{R} \\ \mathrm{P}(Y \mid B, X) &: (T_B, T_X) \to T_Y \to \mathbb{R} \end{aligned}$$

where T_v is the type (domain) of random variable v.

By repeated application of Bayes' theorem, a function *proportional* to the sought probability distribution $\mathrm{P}(A, B \mid X, Y)$ is obtained by the "product" of the distributions of the individual nodes partially applied to the observed data. In our case, given some specific observations x and y of X and Y respectively, we get the two-argument (a and b) functions:

$$\begin{aligned} \mathrm{P}(A, B \mid X, Y)\ (\mathrm{x}, \mathrm{y})\ (a, b) \ \propto\ & \mathrm{P}(Y \mid B, X)\ (b, \mathrm{x})\ \mathrm{y} \\ & \times \mathrm{P}(X \mid A)\ a\ \mathrm{x} \\ & \times \mathrm{P}(B \mid A)\ b\ a \\ & \times \mathrm{P}(A)\ a \end{aligned}$$

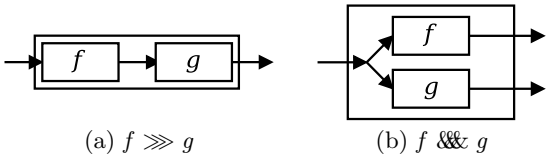
Figure 2 illustrates the situation. The key aspect is that the edges carrying observed data have been reversed, thus "routing" observations to individual nodes in the graph where the given distributions then are partially applied. Given this function that is proportional to the desired distribution, algorithms such as Metropolis-Hastings (a Markov Chain Monte Carlo (MCMC) method) can be used to calculate approximations to the actual distributions of each unknown parameter by repeated sampling. Figure 2 thus represents an *estimation process*.

As should be clear from this example, the model must be a finite network for this type of estimation to workThis means that the *static unfolding*[1]of a probabilistic model (program) for which estimation is desired must be finite.

However, if the notion of computation is a monad, it is easy to construct models that have infinite unfoldings; e.g.

```
foo n = do
  x ← uniform 0 1
  if x < 0.5 then
    foo (n + 1)
  else ...
```

[1]Taking all paths through the code statically; e.g., both the "then" and the "else" branch for a conditional.



(a) $f \ggg g$ (b) $f \mathbin{\&\&\&} g$

Figure 3: Arrow combinators

Here, the static unfolding is an infinite tree. A monad is simply too general as it allows the rest of the computation to depend on the results of computations thus far. As discussed in section 1, this was resolved by making the notion of computation be an *arrow* [3] representing a conditional probability distribution. Figure 3 shows some arrow combinators, and without going into details, it should be intuitively clear how such building blocks can be used to describe networks of the type shown in figure 1.

The central abstraction is the arrow $CP\ o\ a\ b$, where CP stands for "Conditional Probability", a is "the given", b "the outcome", and o observability. Ignoring the observability, an object of type $CP\ o\ T_A\ T_B$ thus represents a conditional probability distribution $\mathrm{P}(B \mid A)$, where T_A is the type of random variable A and T_B the type of variable B.

The observability describes which parts of the given are observable from the outcome; i.e., for which there exists a pure function mapping (part of) the outcome to (part of) the given. Note that observability does *not* mean "will be observed". Observability is determined statically through a type-level computation, and its role is to determine how observations are propagated in the network in "reverse mode", as illustrated in figure 2. However, the observability also imposes static constraints that helps ensuring that the standard estimation methods are applicable.

As an example, the type signature for arrow composition in this setting is:

$$\begin{aligned} (\ggg) :: & \mathit{Fusable}\ o2\ b \\ & \Rightarrow CP\ o1\ a\ b \to CP\ o2\ b\ c \\ & \quad \to CP\ (o1 \ggg o2)\ a\ c \end{aligned}$$

Note that this actually is not a standard arrow, but a constrained, indexed, generalized arrow. Further, the name $\ggg$ has been reused at the type level for the operation that computes the combined observability of two composed arrows.

3. EXAMPLE: THE LIGHTHOUSE

We conclude the paper by implementing the Lighthouse Problem, a classic data analysis problem [9]. A lighthouse is located at some position α along a straight piece of coastline on a rock at a distance β out into the sea: see figure 4. The lighthouse emits flashes that are detected along the shoreline. But due to the nature of the detectors, we only have the positions along the shore at which flashes have been detected, not any information about the directions in which the light flashes were observed. The task is now to determine the position of the lighthouse given a set of positions of detected flashes.

An analysis of the problem shows that the flash positions ought to follow a Cauchy distribution. Interestingly, as the mean of a Cauchy distribution does not exist, naively trying to at least determine the position α along the shore by computing the sample mean will not work.

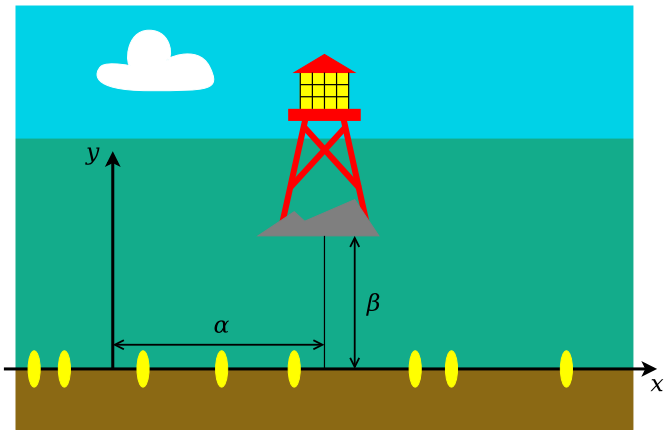


Figure 4: The lighthouse

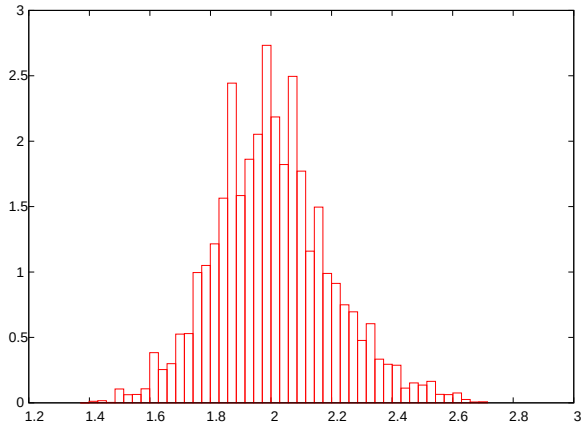


Figure 5: Distribution for parameter β

The following shows how the lighthouse model might be implemented in Ebba:

```
lightHouse :: CP U () [Double]
lightHouse = proc () do
  α ← uniformParam "alpha" (−50) 50 ⤙ ()
  β ← uniformParam "beta" 0 20 ⤙ ()
  xs ← many 10 lightHouseFlash ⤙ (α, β)
  returnA ⤙ xs
```

Our ignorance of the position of the lighthouse is expressed by assuming uniformly distributed priors for the parameters. The distribution *lightHouseFlash* is the aforementioned Cauchy distribution and is implemented separately. The parameter 10 to the combinator *many* is only used in "forward mode" to generate flashes from the model. In backwards, inference, mode, the static number is ignored and the actual number is determined from the number of data points.

The code above uses the so called arrow notation that is available in Haskell. Unfortunately, as described in section 2, the arrow in Ebba is not a standard arrow. One consequence of this is that the arrow notation cannot be used. Thus at present, the model has to be expressed directly in terms of the arrow combinators:

```
lightHouse :: CP U () [Double]
lightHouse = (uniformParam "alpha" (−50) 50
              &&& uniformParam "beta" 0 20)
             ⋙ many 10 lightHouseFlashes
```

However, as this is cumbersome, the aim is to implement support for arrow notation for the Ebba arrow.

To test, a vector of 200 detected flashes was generated at random from the model for $\alpha = 8$ and $\beta = 2$ (the ground truth). The parameter distribution given the outcome was then sampled 100000 times using Metropolis- Hastings (picking every 10th sample from the Markov chain to reduce correlation between samples). Figure 5 shows the resulting distribution for β from one run. While the result is a bit noisy, it is clear that the distance from the shore out to the lighthouse rock has been determined fairly accurately, and that we also get a good idea of how uncertain the result is.

4. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback.

5. REFERENCES

[1] M. Gerdin and J. Sjöberg. Nonlinear stochastic differential-algebraic equations with application to particle filtering. In *Proceedings of the 45th IEEE Conference on Decision & Control*, San Diego, CA, USA, Dec. 2006.

[2] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[3] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[4] L. Ljung. Perspectives on system identification. *Annual Reviews in Control*, 34(1):1–12, Mar. 2010.

[5] D. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS — a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.

[6] T. A. Nielsen. *Baysig Reference Manual*, 2014. `http://www.bayeshive.com`.

[7] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.

[8] H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling from an object-oriented perspective. *Simulation News Europe*, 17(2):29–38, Sept. 2007.

[9] D. S. Sivia and J. Skilling. *Data Analysis: A Bayesian Tutorial*. Oxford University Press, second edition edition, 2006.

[10] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.3*, 2014. `http://mc-stan.org/`.

[11] 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.