Building inference algorithms from monad transformers

Adam Ścibior
University of Cambridge
and MPI Tübingen

Yufei Cai Klaus Ostermann
University of Tübingen

Zoubin Ghahramani
University of Cambridge

Abstract

We show how to decompose popular inference algorithms into a set of simple, reusable building blocks corresponding to monad transformers. We define a collection of such building blocks and implement them in Haskell producing a library for constructing inference algorithms in a modular fashion. We are also working towards formalizing those concepts as monadic denotational semantics for inference algorithms.

Introduction

A common approach to writing denotational semantics for programming languages is to interpret expressions in a particular monad that handles the program's effects. While denotational semantics is usually used for reasoning about programs and justifying correctness of program transformations, it is also possible to build interpreters by directly implementing denotational semantics.

Denotational semantics of probabilistic programs are usually written in some variant of the Giry monad. While convenient for reasoning about programs, the Giry monad is not suitable for a direct implementation, since in general it involves computing intractable integrals. We investigate the possibility of constructing alternative denotational semantics for probabilistic programs that would be suitable for direct implementation.

We build on the approach of Liang et al. (1995), who showed that using monad transformers can lead to a practical and flexible implementation of interpreters by using one layer in the monad stack for each type of effect present in the program. In this work we are investigating how their ideas can be extended to handle probabilistic effects as well.

In principle it would be sufficient to add a single layer to the monad stack corresponding to the Giry monad. This layer would handle all the probabilistic effects, and although there might be some non-trivial interactions between probabilistic and other effects, the task of building an interpreter would be relatively straightforward. Unfortunately this is not feasible in practice.

The task is therefore to build interpreters that execute suitable approximate inference algorithms. However, since there is no one best such algorithm, it is also desirable to have a flexible implementation where the algorithm can be easily adjusted. We show that many popular inference algorithms can be implemented using a stack of suitable monad transformers. Furthermore, alternative variants of such algorithms can often be obtained by simply adding another layer to the monad stack.

We implement a Haskell library[1] for building inference algorithms from monad transformers and use it to obtain simple implementations of popular inference algorithms. The library can either be used for inference on probabilistic programs written directly in Haskell, or as a tool for writing interpreters for other probabilistic programming languages (PPLs). We are also working towards formalizing the relevant concepts in category theory to provide a basis for the construction of monadic denotational semantics of inference algorithms in arbitrary PPLs.

To a certain extent our approach is compatible with the original setup of Liang et al. (1995) in that other effects can be added by introducing more layers to the transformer stack. For example, state can be introduced by adding a state monad transformer on top of the inference transformer stack. We leave for future work the investigation of how other transformers interact with ones used for inference.

Building blocks

We introduce several new terms in this work. The first is a probability monad, which means a monad equipped with operations for drawing random variables from specific distributions and for incorporating likelihood scores. This is very much analogous to an abstract state monad equipped with get and put operations. Additionally we define a probability monad transformer, which is a monad transformer `T` such that for any probability monad `M`, `T M` is a probability monad.

The second term we introduce is inference transformation, which is a natural transformation between two probability monads. An inference algorithm is obtained by composing a sequence of such transformations.

Each building block consists of the following:

- a probability monad transformer `T`
- a set of inference transformations that accept a family of probability monad stacks involving `T`, at least one mapping from `T M` to `M` for all probability monads `M`
- one or more ways to hoist an inference transformation through `T`, that is functions
  `hoist :: (forall a. m a -> n a)-> T m a -> T n a`

Table 1 presents examples of probability monad transformers and Table 2 presents examples of inference transformations associated with them.

In our framework inference algorithms are implemented by first constructing a suitable probability monad transformer stack, then using it to interpret the program, and finally applying a sequence of inference transformations to the result. For example, the Sequential Monte Carlo algorithm can be obtained by applying the following transformation to the interpreted program.

```
smc :: Int -> Int -> Sequential (Population IO) a
                  -> Population IO a
smc k n = flatten .
  composeCopies k (advance . hoist resample) .
  hoist (spawn n >>)
```

[1] `https://github.com/adscib/monad-bayes`

Probability monad	Representation	Haskell type
Discrete distribution	Set of outcome-probability pairs	`Dist a = [(a, Double)]`
Random sampler	Function from entropy source to outcome type	`R a = IO a`
Sample with failure	Distribution on outcomes allowing failure	`Rejection m a = m (Maybe a)`
Weighted sample	Distribution on outcome-weight pairs	`Weighted m a = m (a, Double)`
Population	Distribution on multisets of outcome-weight pairs	`Population m a = m [(a, Double)]`
Suspendable generator	Gradually constructed distributions	`Sequential m a = m (Either a (Sequential m a))`
Joint distribution	Distribution on traces of programs	`Traced m a`

Table 1. A selection of probability monads and probability monad transformers serving as building blocks in our approach to constructing inference algorithms. The list is non-exhaustive but sufficiently rich to construct many of the popular inference algorithms for probabilistic programs. The implementation of `Traced` is too complicated to show here, but essentially it is a list of all random variables and their distributions as used in the program.

Inference operation	Action it performs	Haskell function
Exhaustive enumeration	Sums probabilities over all execution paths	`enumerate :: Dist a -> [(a, Double)]`
Rejection sampling	Attempts to sample, restarts if fails	`rejection :: Rejection m a -> m a`
Increase population	Initialises a population of size `n`	`spawn :: Int -> Population m ()`
Resample population	Draws a new population from the old one	`resample :: Population m a -> Population m a`
Collapse population	Draws a single sample from the population	`collapse :: Population m a -> m a`
Advance generation	Generates the next part of the model	`advance :: Sequential m a -> Sequential m a`
Lightweight Metropolis-Hastings	Randomly mutates the execution trace	`mhStep :: Traced m a -> Traced m a`

Table 2. Example inference transformations acting on probability monads presented in Table 1. While many more can be constructed, this collection is sufficient to build some non-trivial inference algorithms.

where `k` is the number of observations, `n` is the number of particles used, and `composeCopies j :: (a -> b)-> a -> b` applies the given function `j` times.

Our approach is easily extensible. For example, we may want to add Metropolis-Hastings rejuvenation steps after each resampling point, obtaining an algorithm known as resample-move Sequential Monte Carlo. This only requires adding another layer to the transformer stack and some additional hoisting.

```
smcrm :: Int -> Int ->
        Sequential (Traced (Population IO)) a ->
        Population IO a
smcrm k n = marginal . flatten .
  composeCopies k (advance . hoist mhStep .
                     hoist (hoist resample)) .
  (hoist . hoist) (spawn n >>)
```

where one `hoist` is associated with `Traced` and the other with `Sequential`.

Denotational Semantics

Apart from implementing a Haskell library, we are also working on formalizing our approach as a denotational semantics. This would allow us to formally reason about inference transformations and to write practical interpreters for probabilistic programming languages based directly on the formal semantics, potentially even allowing such implementations to be formally verified.

For simple first-order languages this can be done by constructing probability monads in the category **Meas** of measurable spaces, but for higher-order languages this is difficult if not impossible, since **Meas** is not Cartesian closed. The task becomes even more difficult if we add advanced features of modern functional languages such as generalized algebraic datatypes, impredicative polymorphism and higher kinds.

The key to overcoming those difficulties is to observe that countably based Scott domains are separable topological spaces, each of which uniquely defines a measurable space: its standard Borel space. In this way, the category **CDom** of countably based domains can be viewed as a subcategory of **Meas**. Probabilistic programs can be interpreted under the standard domain-theoretic semantics as continuous functions in **CDom**, which are automatically measurable. Scott domains can encode polymorphism, higher-order recursive types and some dependent types. Monads constructed using `Population`, `Sequential` and `Traced` are to be shown as monads over **CDom**, and their probabilistic meaning are expressed through a *monad opfunctor* (Street 1972) to the Giry monad.

Definition 1. Let G : **Meas** $\to$ **Meas** be the Giry monad and let B : **CDom** $\to$ **Meas** be the functor mapping each domain to its standard Borel space. A *probability monad* is a pair (M, ω) such that M is a monad over **CDom** and ω is a natural transformation $BM \to GB$ such that $\omega \circ B\eta_M = \eta_G$ and $\omega \circ B\mu_M = \mu_G \circ G\omega \circ \omega$. If (M, ω_M) and (N, ω_N) are probability monads, then an *exact inference transformation* is a natural transformation $\tau : M \to N$ such that $\tau \circ \omega_M = \omega_N$.

Our goal is to show monads constructed using `Population`, `Sequential` and `Traced` to be probability monads, and the associated inference transformations to be exact. Since exact inference transformations are closed under composition, inference algorithms are exact if constructed from exact building blocks. For importance samplers this means they converge to the true posterior, while for Markov chain Monte Carlo methods it means that they leave the true posterior invariant, a condition known as detailed balance. Definition 1 provides a notion of correctness for inference algorithms, which is satisfied by so-called exact approximate inference methods, like the ones described in this paper, but not by others such as Variational Inference.

References

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. POPL, 1995.

R. Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149 – 168, 1972.