

A compositional approach to scalable Bayesian computation and probabilistic programming

Darren Wilkinson

@darrenjw

darrenjw.github.io

Newcastle University, UK / The Alan Turing Institute

Advances and challenges in machine learning languages

Cambridge University

20-21 May 2019

Overview

Compositionality, category theory, and functional programming

- Compositionality

- Functional Programming

- Category Theory

Probability monads

- Composing random variables

- Implementations of probability monads

- Probabilistic programming

Summary and conclusions

Compositionality, category theory, and functional programming

Compositionality and modelling

- We typically solve big problems by (recursively) breaking them down into smaller problems that we can solve more easily, and then **compose** the solutions of the smaller problems to provide a solution to the big problem that we are really interested in
- This “**divide and conquer**” approach is necessary for the development of genuinely **scalable** models and algorithms
- Statistical models and algorithms are not usually formulated in a composable way
- **Category theory** is in many ways the **mathematical study of composition**, and provides significant insight into the development of more compositional models of computation

What is functional programming?

- FP languages emphasise the use of **immutable** data, **pure**, **referentially transparent functions**, and **higher-order functions**
- Unlike commonly used **imperative** programming languages, they are closer to the Church end of the **Church-Turing thesis** — eg. closer to **Lambda-calculus** than a **Turing-machine**
- The original Lambda-calculus was **untyped**, corresponding to a **dynamically-typed** programming language, such as **Lisp**
- **Statically-typed** FP languages (such as **Haskell**) are arguably more scalable, corresponding to the **simply-typed Lambda-calculus**, closely related to **Cartesian closed categories**...

Monadic collections (in Scala)

- A collection of type `M[T]` can contain (multiple) values of type `T`
- If the collection supports a higher-order function `map(f: T =>S): M[S]` then we call the collection a **Functor**
 - eg. `List(1,3,5,7) map (x =>x*2) = List(2,6,10,14)`
- If the collection additionally supports a higher-order function `flatMap(f: T =>M[S]): M[S]` then we call the collection a **Monad**
 - eg. `List(1,3,5,7) flatMap (x =>List(x,x+1)) = List(1, 2, 3, 4, 5, 6, 7, 8)`
 - instead of `List(1,3,5,7) map (x =>List(x,x+1)) = List(List(1,2),List(3,4),List(5,6),List(7,8))`

Composing monadic functions

- Given functions $f: S \Rightarrow T$, $g: T \Rightarrow U$, $h: U \Rightarrow V$, we can compose them as `h compose g compose f` or `s =>h(g(f(s)))` to get $hgf: S \Rightarrow V$
- Monadic functions $f: S \Rightarrow M[T]$, $g: T \Rightarrow M[U]$, $h: U \Rightarrow M[V]$ don't compose directly, but do using `flatMap`:
`s =>f(s) flatMap g flatMap h` has type $S \Rightarrow M[V]$
- Can be written as a **for-comprehension** (**do** in Haskell):
`s =>for (t<-f(s); u<-g(t); v<-h(u)) yield v`
- Just syntactic sugar for the chained `flatMap`s above — really **not** an imperative-style “for loop” at all...

Other monadic types: Prob/Gen/Rand

- The **Probability monad** is another important monad with obvious relevance to statistical computing
- A `Rand[T]` represents a random quantity of type `T`
- It is used to encapsulate the non-determinism of functions returning random quantities — otherwise these would break the **purity** and **referential transparency** of the function
- `map` is used to transform one random quantity into another
- `flatMap` is used to chain together stochastic functions to create joint and/or marginal random variables, or to **propagate uncertainty** through a computational work-flow or pipeline
- Probability monads form the basis for the development of **probabilistic programming languages** using FP

Category theory

- A category \mathcal{C} consists of a collection of **objects**, $\text{ob}(\mathcal{C})$, and **morphisms**, $\text{hom}(\mathcal{C})$. Each morphism is an ordered pair of objects (an arrow between objects). For $x, y \in \text{ob}(\mathcal{C})$, the set of morphisms from x to y is denoted $\text{hom}_{\mathcal{C}}(x, y)$.
 $f \in \text{hom}_{\mathcal{C}}(x, y)$ is often written $f : x \longrightarrow y$.
- Morphisms are closed under **composition**, so that if $f : x \longrightarrow y$ and $g : y \longrightarrow z$, then there must also exist a morphism $h : x \longrightarrow z$ written $h = g \circ f$.
- Composition is associative, so that $f \circ (g \circ h) = (f \circ g) \circ h$ for all composable $f, g, h \in \text{hom}(\mathcal{C})$.
- For every $x \in \text{ob}(\mathcal{C})$ there exists an **identity** morphism $\text{id}_x : x \longrightarrow x$, with the property that for any $f : x \longrightarrow y$ we have $f = f \circ \text{id}_x = \text{id}_y \circ f$.

Examples of categories

- The category **Set** has an object for every **set**, and its morphisms represent set **functions**
 - Note that this is a category, since functions are composable and we have identity functions, and function composition is associative
 - Note that objects are “atomic” in category theory — it is not possible to “look inside” the objects to see the set elements — category theory is “point-free”
- For a pure FP language, we can form a category where objects represent **types**, and morphisms represent **functions** from one type to another
 - In Haskell this category is often referred to as **Hask**
 - This category is very similar to **Set**, in practice (both CCCs)
 - By modelling FP types and functions as a category, we can bring ideas and techniques from CT into FP

Functors

- A **functor** is a mapping from one category to another which preserves some structure
- A functor F from \mathcal{C} to \mathcal{D} , written $F : \mathcal{C} \longrightarrow \mathcal{D}$ is a pair of functions (both denoted F):
 - $F : \text{ob}(\mathcal{C}) \longrightarrow \text{ob}(\mathcal{D})$
 - $F : \text{hom}(\mathcal{C}) \longrightarrow \text{hom}(\mathcal{D})$, where $\forall f \in \text{hom}(\mathcal{C})$, we have $F(f : x \longrightarrow y) : F(x) \longrightarrow F(y)$
 - In other words, if $f \in \text{hom}_{\mathcal{C}}(x, y)$, then $F(f) \in \text{hom}_{\mathcal{D}}(F(x), F(y))$
- The functor must satisfy the **functor laws**:
 - $F(\text{id}_x) = \text{id}_{F(x)}, \forall x \in \text{ob}(\mathcal{C})$
 - $F(f \circ g) = F(f) \circ F(g)$ for all composable $f, g \in \text{hom}(\mathcal{C})$
- A functor $F : \mathcal{C} \longrightarrow \mathcal{C}$ is called an **endofunctor** — in the context of functional programming, the word functor usually refers to an endofunctor $F : \mathbf{Set} \longrightarrow \mathbf{Set}$

Natural transformations

- Often there are multiple functors between pairs of categories, and sometimes it is useful to be able to transform one to another
- Suppose we have two functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$
- A **natural transformation** $\alpha : F \Rightarrow G$ is a family of morphisms in \mathcal{D} , where $\forall x \in \mathcal{C}$, the **component** $\alpha_x : F(x) \longrightarrow G(x)$ is a morphism in \mathcal{D}
- To be considered **natural**, this family of morphisms must satisfy the **naturality law**:
 - $\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \longrightarrow y \in \text{hom}(\mathcal{C})$
- **Naturality** is one of the most fundamental concepts in category theory
- In the context of FP, a natural transformation could (say) map an **Option** to a **List** (with at most one element)

Monads

- A **monad** on a category \mathcal{C} is an endofunctor $T : \mathcal{C} \rightarrow \mathcal{C}$ together with two natural transformations $\eta : \text{Id}_{\mathcal{C}} \rightarrow T$ (**unit**) and $\mu : T^2 \rightarrow T$ (**multiplication**) fulfilling the **monad laws**:
 - **Associativity**: $\mu \circ T\mu = \mu \circ \mu_T$, as transformations $T^3 \rightarrow T$
 - **Identity**: $\mu \circ T\eta = \mu \circ \eta_T = 1_T$, as transformations $T \rightarrow T$
- The associativity law says that the two ways of **flattening** $T(T(T(x)))$ to $T(x)$ are the same
- The identity law says that the two ways of **lifting** $T(x)$ to $T(T(x))$ and then flattening back to $T(x)$ both get back to the original $T(x)$
- In FP, we often use **M** (for monad) rather than T (for triple), and say that there are three monad laws — the identity law is considered to be two separate laws

Kleisli category

- Kleisli categories formalise monadic composition
- For any monad T over a category \mathcal{C} , the **Kleisli category** of \mathcal{C} , written \mathcal{C}_T is a category with the same objects as \mathcal{C} , but with morphisms given by:
 - $\text{hom}_{\mathcal{C}_T}(x, y) = \text{hom}_{\mathcal{C}}(x, T(y)), \forall x, y \in \text{ob}(\mathcal{C})$
- The identity morphisms in \mathcal{C}_T are given by $\text{id}_x = \eta(x), \forall x$, and morphisms $f : x \rightarrow T(y)$ and $g : y \rightarrow T(z)$ in \mathcal{C} can compose to form $g \circ_T f : x \rightarrow T(z)$ via
 - $g \circ_T f = \mu_z \circ T(g) \circ f$leading to composition of morphisms in \mathcal{C}_T .
- In FP, the morphisms in \mathcal{C}_T are often referred to as **Kleisli arrows**, or **Kleislis**, or sometimes just **arrows** (although **Arrow** usually refers to a generalisation of Kleisli arrows, sometimes known as **Hughes arrows**)

Comonads

- The comonad is the categorical dual of the monad, obtained by “reversing the arrows” for the definition of a monad
- A **comonad** on a category \mathcal{C} is an endofunctor $W : \mathcal{C} \rightarrow \mathcal{C}$ together with two natural transformations $\varepsilon : W \rightarrow \text{Id}_{\mathcal{C}}$ (**counit**) and $\delta : W \rightarrow W^2$ (**comultiplication**) fulfilling the **comonad laws**:
 - **Associativity**: $\delta_W \circ \delta = W\delta \circ \delta$, as transformations $W \rightarrow W^3$
 - **Identity**: $\varepsilon_W \circ \delta = W\varepsilon \circ \delta = 1_W$, as transformations $W \rightarrow W$
- The associativity law says that the two ways of **duplicating** a $W(x)$ duplicated to a $W(W(x))$ to a $W(W(W(x)))$ are the same
- The identity law says that the two ways of **extracting** a $W(x)$ from a $W(x)$ duplicated to a $W(W(x))$ are the same

Typeclasses

- **Typeclasses** are a mechanism for supporting **ad hoc polymorphism** in (functional) programming languages
- They are more flexible way to provide polymorphic functionality than traditional inheritance-based object classes in conventional object-oriented programming languages
- To define a typeclass (such as **Monoid**) for a basic type, the language must support **parametric types**
- To define a typeclass (such as **Functor** or **Monad**) for a parametric type or type constructor, the language must support **higher-kinded types** (very few widely-used languages do)

Typeclasses for Monoid, Functor, Monad and Comonad

```
trait Monoid[A] {  
  def combine(a1: A, a2: A): A  
  def id: A  
}  
  
trait Functor[F[_]] {  
  def map[A,B](fa: F[A])(f: A => B): F[B]  
}  
  
trait Monad[M[_]] extends Functor[M] {  
  def pure[A](a: A): M[A]  
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]  
}  
  
trait Comonad[W[_]] extends Functor[W] {  
  def extract[A](wa: W[A]): A  
  def coflatMap[A,B](wa: W[A])(f: W[A] => B): W[B]  
}
```

Comonads for statistical computation

- Monads are good for operations that can be carried out on data points independently
- For computations requiring knowledge of some kind of local neighbourhood structure, **Comonads** are a better fit
- `coflatMap` will take a function representing a **local computation** producing one value for the new structure, and then extend this to generate all values associated with the comonad
- Useful for defining linear filters, Gibbs samplers, convolutional neural networks, etc.

Probability monads

Composing random variables with the probability monad

- The **probability monad** provides a foundation for describing random variables in a pure functional way (cf. **Giry monad**)
- We can build up joint distributions from marginal and conditional distributions using **monadic composition**
- For example, consider an exponential mixture of Poissons (marginally negative binomial): we can think of an exponential distribution parametrised by a rate as a function `Exponential: Double => Rand[Double]` and a Poisson parametrised by its mean as a function `Poisson: Double => Rand[Int]`
- Those two functions don't directly compose, but do in the Kleisli category of the `Rand` monad, so `Exponential(3) flatMap {Poisson(_)}` will return a `Rand[Int]` which we can draw samples from if required

Monads for probabilistic programming

- For larger probability models we can use **for-comprehensions** to simplify the model building process, eg.

```
for { mu <- Gaussian(10,1)
      tau <- Gamma(1,1)
      sig = 1.0/sqrt(tau)
      obs <- Gaussian(mu,sig) }
yield ((mu,tau,obs))
```

- We can use a regular probability monad for building forward models this way, and even for building models with simple Bayesian inference procedures allowing conditioning
- For sophisticated probabilistic sampling algorithms (eg. SMC, MCMC, pMCMC, HMC, ...) and hybrid compositions, it is better to build models like this using a **free monad** which can be **interpreted** in different ways

Probability monad foundations

Mathematically, what **exactly** is a probability monad P ?

Standard answer:

- **Giry monad** – measurable functions and spaces
 - Defined on the category **Meas** of measurable spaces, P sends X to the space of probability measures on X
 - η is a dirac measure ($\eta(x) = \delta_x$) and μ is defined as marginalisation using Lebesgue integration

$$\mu_X(\rho)(A) = \int_{P(X)} \tau_A(d\rho)$$

- Provides a solid foundation matching up closely with conventional probability theory, but isn't as compositional as we'd like (eg. **Meas** is not cartesian closed)
- Awkward for (higher-order) probabilistic programming languages

Alternative probability monad foundations

- **Quasi-Borel spaces**
 - A modification of the measure space approach which is cartesian closed (eg. $\mathbb{R}^{\mathbb{R}} = \mathbf{QBS}(\mathbb{R}, \mathbb{R})$)
 - Good for the denotational semantics of (higher-order) probabilistic programming languages where we want to define probability distributions over functions
- **Kantorovich monad** – built on (complete) metric spaces
 - Provides an alternative, more composable foundation for probability theory, less tightly linked to measure theory
- **Expectation monad** – directly define an expectation monad
 - A formulation in which expectation is primitive and probability is a derived concept (cf. de Finetti)

And several others...

Rand for forward simulation

- Whilst the semantics of probability monads should be reasonably clear, there are many different ways to implement them, depending on the intended use-case
- The simplest probability monad implementations typically provide a `draw` method, which can be used (with a uniform random number generator) to generate draws from the distribution of interest
- For `x: Rand[X]`, monadic bind `x flatMap (f: X => Rand[Y])` returns `y: Rand[Y]`
- `f` represents a conditional distribution and `y` represents the marginalisation of this distribution over `x`
- The `draw` method for `y` first calls `draw` on `x` (which it holds a reference to), feeds this in to `f` and then calls `draw` on the result

Dist for Bayesian conditioning via SMC

- Rather than providing a `draw` method for generating individual values from the distribution of interest, you can define a monad whose values represent large (weighted) random samples from a distribution — an empirical distribution
- `flatMap` is then essentially just the same `flatMap` you would have on any other collection, but here will typically be combined with a random thinning of the result set to prevent an explosion in the number of particles with deep chaining
- One advantage of this representation is that it then easy to introduction a `condition` method which uses importance (re)sampling to condition on observations
- This can be used to implement a simple SMC-based Bayesian PPL with very little code, but it won't scale well with large or complex models

RandomVariable for Bayesian HMC sampling

- Rather than using a probability monad to represent samples or methods for sampling, one can instead use them to represent the (joint, log) density of the variables
- `flatMap` just multiplies the (conditional) densities
- Again, conditioning is easy (multiplication), so this forms a good basis for Bayesian PPLs
- Can use the joint posterior for simple MH algorithms (and Gibbs, if dependencies are tracked), but for Langevin and HMC algorithms, also need to keep track of gradients, using automatic differentiation (AD)
- OK, because (reverse-mode) AD on a compute graph is also monadic!
- `Rainier` is a Scala library for HMC sampling of monadic random variables (using a static compute graph, for efficiency)

Example — Bayesian logistic regression model in Rainier

```
val model = for {
  beta0 <- Normal(0, 5).param
  beta1 <- Normal(0, 5).param
  _ <- Predictor.fromDouble { x =>
    {
      val theta = beta0 + beta1 * x
      val p = Real(1.0) / (Real(1.0) +
        (Real(0.0) - theta).exp)
      Categorical.boolean(p)
    }
  }.fit(x zip y)
} yield Map("b0" -> beta0, "b1" -> beta1)

val out = model.sample(HMC(5), 1000, 10000*10, 10)
```

Composing probabilistic programs

- Describing probabilistic programs as **monadic values** in a functional programming language with syntax for monadic composition leads immediately to an **embedded PPL DSL** “for free”
- This in turn enables a fully **compositional** approach to the (scalable) development of (hierarchical) probabilistic models
- Model components can be easily “re-used” in order to build big models from small
- eg. a regression model component can be re-used to create a hierarchical random effects model over related regressions
- In some well-known PPLs (eg. BUGS), this would require manual copy-and-pasting of code, wrapping with a “for loop”, and manual hacking in of an extra array index into all array references — not at all compositional!

Representation independence using the free monad

- However you implement your probability monad, the semantics of your probabilistic program are (essentially) the same
- It would be nice to be able to define and compose probabilistic programs independently of concerns about implementation, and then to **interpret** the program with a particular implementation later
- Building a probability monad on top of the **free monad** allows this — implementation of **pure** and **flatMap** is “suspended” in a way that allows subsequent interpretation with concrete implementations later
- This allows **layering** of multiple inference algorithms, and different interpretation of different parts of the model, enabling sophisticated **composition** of different (hybrid) inference algorithms

Compositionality of inference algorithms







- As well as building **models** in scalable, compositional way, we would also like our **inference algorithms** to be compositional, ideally reflecting the compositional structure of our models
- Some algorithms, such as component-wise samplers and message-passing algorithms, naturally reflect the compositional structure of the underlying model
- Other algorithms, such as Langevin and HMC samplers, deliberately don't decompose with the model structure, but do have other structure that can be exploited, such as decomposing over observations
- Understanding and exploiting the **compositional structure** of models and algorithms will be crucial for developing scalable inferential methods

Summary and conclusions

Summary

- Mathematicians and theoretical computer scientists have been thinking about models of (scalable) computation for decades
- **Functional programming languages** based on Cartesian closed categories provide a sensible foundation for computational modelling with appropriate levels of abstraction
- Concepts from **category theory**, such as functors, monads and comonads, provide an appropriate array of tools for scalable data modelling and algorithm construction and composition
- Expressing models and algorithms in FP languages using category theory abstractions leads to **elegant, composable PPLs “for free”**, doing away with the need to manually construct and parse custom DSLs
- Monadic composition is **the canonical** approach to constructing flexible, composable PPLs (and AD systems, and deep NNs, ...)

References

-  Fong, B., Spivak, D., Tuyéras, R. (2019) **Backprop as Functor: A compositional perspective on supervised learning**, *arXiv*, 1711.10455
-  Heunen, C., Kammar, O., Staton, S., Yang, H. (2017) **A convenient category for higher-order probability theory**, *32nd ACM/IEEE LICS.*, 1–12.
-  Law, J., Wilkinson, D. J. (2018) **Composable models for online Bayesian analysis of streaming data**, *Statistics and Computing*, **28**(6):1119–1137.
-  Park, S., Pfenning, F., Thrun, S. (2008) **A probabilistic language based on sampling functions**, *TOPLAS*, **31**(1):4.
-  Ścibior, A., Ghahramani, Z. and Gordon, A. D. (2015) **Practical probabilistic programming with monads**, *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 165–176.
-  Ścibior, A., Kammar, O., Ghahramani, Z. (2018) **Functional programming for modular Bayesian inference**, *Proc. ACM Prog. Lang.*, **2**(ICFP): 83.

Rainier: github.com/stripe/rainier

darrenjw.github.io

@darrenjw