An introduction to category theory and functional programming for scalable statistical modelling and computation

Darren Wilkinson
@darrenjw
tinyurl.com/darrenjw
School of Mathematics & Statistics
Newcastle University, UK

Statistics Seminar, Newcastle University
3rd February 2017

Talk outline

- What's wrong with the current state of statistical modelling and computation?
- What is functional programming (FP) and why is it better than conventional imperative programming?
- What is category theory (CT) and what has it got to do with FP?
- How can we use CT and FP to make statistical computing more scalable?
- What does "scalable" mean, anyway?
- Some examples along the way...

What's up with statistical computing?

- Everything!
- R has become the de facto standard programming language for statistical computing — the S language was designed by statisticians for statisticians in the mid 1970's, and it shows!
  - Many dubious language design choices, meaning it will always be ugly, slow and inefficient (without many significant breaking changes to the language)
  - R's inherent inefficiencies mean that much of the R code-base isn't in R at all, but instead in other languages, such as Fortran, C and C++
  - Although faster and more efficient than R, these languages are actually all even worse languages for statistical computing than R!

Pre-historic programming languages

- The fundamental problem is that all of the programming languages commonly used for scientific and statistical computing were designed 30-50 years ago, in the dawn of the computing age, and haven't significantly changed
  - Think how much computing hardware has changed in the last 40 years!
  - But the language you are using was designed for that hardware using the knowledge of programming languages that existed at that time
  - Think about how much statistical methodology has changed in the last 40 years — you wouldn't use 40 year old methodology — why use 40 year old languages to implement it?!

Modern programming language design

- We have learned just as much about programming and programming languages in the last 40 years as we have about everything else
- Our understanding has developed in parallel with developments in hardware
- People have been thinking a lot about how languages can and should exploit modern computing hardware such as multi-core processors and parallel computing clusters
- Modern functional programming languages are emerging as better suited to modern hardware

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...

Functional programming

- In pure FP, all state is immutable — you can assign names to things, but you can't change what the name points to — no "variables" in the usual sense
- Functions are pure and referentially transparent — they can't have side-effects — they are just like functions in mathematics...
- Functions can be recursive, and recursion can be used to iterate over recursive data structures — useful since no conventional "for" or "while" loops in pure FP languages
- Functions are first class objects, and higher-order functions (HOFs) are used extensively — functions which return a function or accept a function as argument

Concurrency, parallel programming and shared mutable state

- Modern computer architectures have processors with several cores, and possibly several processors
- Parallel programming is required to properly exploit this hardware
- The main difficulties with parallel and concurrent programming using imperative languages all relate to issues associated with shared mutable state
- In pure FP, state is not mutable, so there is no mutable state, and hence no shared mutable state
- Most of the difficulties associated with parallel and concurrent programming just don't exist in FP — this has been one of the main reasons for the recent resurgence of FP languages

Ideal languages for statistical computing

- We should approach the problem of statistical modelling and efficient computation in a modular, composable, functional way
- To do this we need programming languages which are:
  - Strongly statically typed (but with type inference)
  - Compiled (but possibly to a VM)
  - Functional (with support for immutable values, immutable collections, ADTs and higher-order functions)
  - and have support for typeclasses and higher-kinded types, allowing the adoption of design patterns from category theory
- For efficient statistical computing, it can be argued that evaluation should be strict rather than lazy by default
- Scala is a popular language which meets the above constraints

Monadic collections

- 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))`

Other monadic types: Option

- Some computations can fail, and we can capture that possibility with a type called `Option`
  - in Scala — it is `Optional` in Java 8 and `Maybe` in Haskell
- An `Option[T]` can contain `Some[T]` or `None`
- So if we have `chol: Matrix =>Option[TriMatrix]` we can check to see if we have a result
- But if we also have `triSolve: (TriMatrix,Vector) =>Option[Vector]`, how do we "compose" these?
  - `chol(mat) map (tm =>triSolve(tm,vec))` has type `Option[Option[Vector]]` which isn't quite what we want
  - `chol(mat) flatMap (tm =>triSolve(tm,vec))` has type `Option[Vector]` which we do want
  - `flatMap` allows composition of monadic functions

Composing monadic functions

- Given functions `f: S =>T`, `g: T =>U`, `h: U =>V`, we can compose them as `h compose g compose f` or `s =>h(g(f(s)))` to get `hgf: S =>V`
- Monadic functions `f: S =>M[T]`, `g: T =>M[U]`, `h: U =>M[V]` don't compose directly, but do using `flatMap`: `s =>f(s) flatMap g flatMap h` has type `S =>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: Future

- A `Future[T]` is used to dispatch a (long-running) computation to another thread to run in parallel with the main thread
- When a `Future` is created, the call returns immediately, and the main thread continues, allowing the `Future` to be "used" before its result (of type `T`) is computed
- `map` can be used to transform the result of a `Future`, and `flatMap` can be used to chain together `Futures` by allowing the output of one `Future` to be used as the input to another
- `Future`s can be transformed using `map` and `flatMap` irrespective of whether or not the `Future` computation has yet completed and actually contains a value
- `Future`s are a powerful method for developing parallel and concurrent programs in a modular, composable way

Other monadic types: Prob/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
- The probability monad is typically implemented as a State monad, the mechanism for handling mutable state using FP

Parallel monadic collections

- Using `map` to apply a pure function to all of the elements in a collection can clearly be done in parallel
- So if the collection contains n elements, then the computation time can be reduced from $O(n)$ to $O(1)$ (on infinite parallel hardware)
  - `Vector(3,5,7) map (_*2) = Vector(6,10,14)`
  - `Vector(3,5,7).par map (_*2) = ParVector(6,10,14)`
- We can carry out reductions as folds over collections: `Vector(6,10,14).par reduce (_+_) = 30`
- In general, sequential folds can not be parallelised, but...

Monoids and parallel "map–reduce"

- A monoid is a very important concept in FP
- For now we will think of a monoid as a set of elements with a binary relation $\star$ which is closed and associative, and having an identity element wrt the binary relation
- You can think of it as a semi-group with an identity or a group without an inverse
- `fold`s, `scan`s and `reduce` operations can be computed in parallel using tree reduction, reducing time from $O(n)$ to $O(\log n)$ (on infinite parallel hardware)
- "map–reduce" is just the pattern of processing large amounts of data in an immutable collection by first mapping the data (in parallel) into a monoid and then tree-reducing the result (in parallel)

Category theory

- A category $\mathcal{C}$ consists of a collection of objects, $\mathrm{ob}(\mathcal{C})$, and morphisms, $\hom(\mathcal{C})$. Each morphism is an ordered pair of objects (an arrow between objects). For $x, y \in \mathrm{ob}(\mathcal{C})$, the set of morphisms from x to y is denoted $\hom_{\mathcal{C}}(x, y)$. $f \in \hom_{\mathcal{C}}(x, y)$ 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 $f, g, h \in \hom(\mathcal{C})$.
- For every $x \in \mathrm{ob}(\mathcal{C})$ there exists an identity morphism $\mathrm{id}_x : x \longrightarrow x$, with the property that for any $f : x \longrightarrow y$ we have $f = f \circ \mathrm{id}_x = \mathrm{id}_y \circ f$.

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

Set and **Hask**

- $0 \in \mathrm{ob}(\textbf{Set})$ is the empty set, $\emptyset$
  - There is a unique morphism from 0 to every other object — it is an example of the concept of an initial object
  - 0 in **Set** corresponds to the type `Void` in **Hask**, the type with no values
- $1 \in \mathrm{ob}(\textbf{Set})$ is a set containing exactly one element (and all such objects are isomorphic)
  - There is a unique morphism from every other object to 1 — it is an example of the concept of a terminal object
  - 1 in **Set** corresponds to the type `Unit` in **Hask**, the type with exactly one value, `()`
  - Morphisms from 1 to other objects must represent constant functions, and hence must correspond to elements of a set or values of a type — so we can use morphisms from 1 to "look inside" our objects if we must...

Monoid as a category with one object

- Given our definition of a category, we can now reconsider the notion of a monoid now as a category with one object
- The object represents the "type" of the monoid, and the morphisms represent the "values"
- From our definition of a category, we know that there is an identity morphism, that the morphisms are closed under composition, and that they are associative...
- For a monoid type object, M in **Hask**, the (endo)morphisms represent functions, $f_a : M \longrightarrow M$ defined by $f_a(m) = m \star a$
- Again, we see that it is the morphisms that really matter, and that these can be used to "probe" the "internal structure" of an object...

Functors

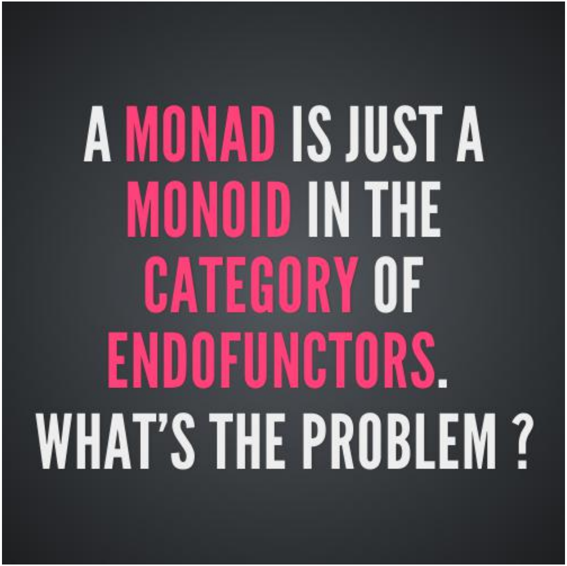
- A functor is a mapping from one category to another which preserves some structure
- A functor F from $\mathcal{C}$ to $\mathcal{D}$, written $F : \mathcal{C} \longrightarrow \mathcal{D}$ is a pair of functions (both denoted F):
  - $F : \mathrm{ob}(\mathcal{C}) \longrightarrow \mathrm{ob}(\mathcal{D})$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

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:
  - $\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \longrightarrow y \in \hom(\mathcal{C})$
- 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 $T : \mathcal{C} \longrightarrow \mathcal{C}$ together with two natural transformations $\eta : \mathrm{Id}_{\mathcal{C}} \longrightarrow T$ (unit) and $\mu : T^2 \longrightarrow T$ (multiplication) fulfilling the monad laws:
  - Associativity: $\mu \circ T\mu = \mu \circ \mu_T$, as transformations $T^3 \longrightarrow T$
  - Identity: $\mu \circ T\eta = \mu \circ \eta_T = 1_T$, as transformations $T \longrightarrow T$
- 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 additional law corresponds to the naturality of μ


A MONAD IS JUST A
MONOID IN THE
CATEGORY OF
ENDOFUNCTORS.
WHAT'S THE PROBLEM ?

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:
  - $\hom_{\mathcal{C}_T}(x,y) = \hom_{\mathcal{C}}(x, T(y)),\ \forall x,y \in \mathrm{ob}(\mathcal{C})$
- The identity morphisms in $\mathcal{C}_T$ are given by $\mathrm{id}_x = \eta(x), \forall x$, and morphisms $f : x \longrightarrow T(y)$ and $g : y \longrightarrow T(z)$ in $\mathcal{C}$ can compose to form $g \circ_T f : x \longrightarrow T(z)$ via
  - $g \ \circ_T f = \mu_z \circ T(g) \circ f$

  leading to composition of morphisms in $\mathcal{C}_\mathcal{T}$.
- In FP, the morphisms in 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)


SAY MONAD
ONE MORE TIME
memegenerator.net

Apache Spark

- We have already seen how parallel monadic collections can automatically parallelise "map" and "reduce" operations
- Apache Spark is a Scala library for Big Data analytics on (large) clusters of machines (in the cloud)
- The basic datatype provided by Spark is an RDD — a resilient distributed dataset
- An RDD is just a lazy, distributed, parallel monadic collection, supporting methods such as `map`, `flatMap`, `reduce`, etc., which can be used in exactly the same way as any other monadic collection
- Code looks exactly the same whether the RDD is a small dataset on a laptop or terabytes in size, distributed over a large Spark cluster

Laziness, composition, laws and optimisations

- Laziness allows some optimisations to be performed that would be difficult to automate otherwise
- Consider a dataset `rdd: RDD[T]`, functions `f: T =>U`, `g: U =>V`, and a binary operation `op: (V,V) =>V` for monoidal type `V`
- We can map the two functions and then reduce with:
  - `rdd map f map g reduce op`
  - to get a value of type `V`, all computed in parallel
- However, re-writing this as:
  - `rdd map (g compose f) reduce op`
  - would eliminate an intermediate collection, but is equivalent due to the 2nd functor law
- Category theory laws often correspond to optimisations that can be applied to code without affecting results — Spark can do these optimisations automatically due to lazy evaluation

Distributed computation

- Big data frameworks such as Spark have been developed for the analysis of huge (internet scale) datasets on large clusters in the cloud
- They typically work by layering on top of a distributed file system (such as HDFS) which distributes a data set across a cluster and leaves data in place, sending required computation across the network to the data
- With a little thought, it is clear that even in the case of "small data" but "big models"/"big computation", these frameworks can be exploited for distributing computation

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 and Monad

- In Scala, we can define typeclasses for `Monoid`, `Functor` and `Monad` (using parametric and higher-kinded types):

```
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 unit[A](a: A): M[A]
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]
}
```

A generic collection typeclass

- We can define a typeclass for generic monadic collections:

```
trait GenericColl[C[_]] {
  def map[A,B](ca: C[A])(f: A => B): C[B]
  def reduce[A](ca: C[A])(f: (A, A) => A): A
  def flatMap[A,B,D[B] <: GenTraversable[B]](
                  ca: C[A])(f: A => D[B]): C[B]
  def zip[A,B](ca: C[A])(cb: C[B]): C[(A, B)]
  def length[A](ca: C[A]): Int
}
```

  and then define instances for standard collections (eg. `Vector`), parallel collections (eg. `ParVector`), and distributed parallel collections (eg. `RDD`)
- We can then write code that is completely parallelisation–agnostic

A scalable particle filter

Single–observation update of a bootstrap particle filter:

```
def update[S: State, O: Observation,
                            C[_]: GenericColl](
    dataLik: (S, O) => LogLik, stepFun: S => S
       )(x: C[S], o: O): (LogLik, C[S]) = {
  val xp = x map (stepFun(_))
  val lw = xp map (dataLik(_, o))
  val max = lw reduce (math.max(_, _))
  val rw = lw map (lwi => math.exp(lwi - max))
  val srw = rw reduce (_ + _)
  val l = rw.length
  val z = rw zip xp
  val rx = z flatMap (p => Vector.fill(
             Poisson(p._1 * l / srw).draw)(p._2))
  (max + math.log(srw / l), rx)
}
```

Filtering as a functional fold

- Once we have a function for executing one step of a particle filter, we can produce a function for particle filtering as a functional fold over a sequence of observations:

```
def pFilter[S: State, O: Observation,
    C[_]: GenericColl, D[O] <: GenTraversable[O]](
    x0: C[S], data: D[O], dataLik: (S, O) => LogLik,
    stepFun: S => S ): (LogLik, C[S]) = {
  val updater = update[S, O, C](dataLik, stepFun) _
  data.foldLeft((0.0, x0))((prev, o) => {
    val next = updater(prev._2, o)
    (prev._1 + next._1, next._2)
  })
}
```

Again, completely parallelisation–agnostic...

Scalable statistical modelling

- We have looked a lot at scalable statistical computation, but what about scalable statistical modelling more generally?
- Independently of any computational issues, statistical modelling of large, complex problems is all about structure, modularity and composition — again, the domain of category theory...
- When Bayesian hierarchical modelling, we often use probabilistic programming languages (such as BUGS, JAGS, Stan...) to build up a large, complex (DAG) model from simple components
- It turns out that monads, and especially free monads, can give us a different (better?) perspective on building and inferring probabilistic models

Composing random variables with the probability monad

- The probability monad provides a foundation for describing random variables in a pure functional way
- 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, ...) it is better to build models like this using a free monad which can be interpreted in different ways

Summary

- You can't learn much about either FP or CT in a single talk/seminar
- I don't expect everyone to have understood everything!
- The aim was to give a little insight into:
  - Why FP is interesting, and inherently more modular, composable and scalable than imperative programming
  - Why CT is a good model for composable computation (because it is a theory of structure and composition)
  - Why CT provides powerful abstractions which make FP easier, more modular, and more general

Conclusions

- We should approach the problem of statistical modelling and computation in a modular, composable, functional way, guided by underpinning principles from category theory
- To implement solutions to problems in statistical modelling and computation in a more scalable way, we need programming languages which are:
  - Strongly statically typed
  - Compiled
  - Functional
  - and support typeclasses and higher-kinded types
- Scala and Spark provide a nice illustration of the power of this approach, but there are other interesting languages, including: Haskell, (S)ML, OCaml, Frege, Eta, ...
- For more about Scala: `darrenjw.wordpress.com`