A more functional future for statistical computing?

Darren Wilkinson
@darrenjw
tinyurl.com/darrenjw
Newcastle University, UK

and

The Alan Turing Institute

APTS Statistical Computing

Cambridge University

14th December 2018
Background
Changing face of statistics

- The practice of academic statistics, and especially computational statistics, has changed beyond recognition in the last 40 years
- It is likely to change even more in the next 40 years
- This is the age of data...
- Boundaries are blurring between statistics and related disciplines, such as machine learning, AI, and data science
- Computing and computational approaches are progressing very rapidly in all of these disciplines, and this is impacting how we do and think about computing in statistics
- Most people in these other fields don’t use R
- Will we still be using R in 20 years time? I doubt it, and I really hope not!
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

• 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
  • Compare with 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?!
Compositionality, category theory, and functional programming
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.
Compositionality and programming

- The programming languages typically used for scientific and statistical computing also fail to naturally support composition of models, data and computation.
- Functional programming languages which are strongly influenced by category theory turn out to be much better suited to the development of scalable statistical algorithms than the imperative programming languages more commonly used.
- Expressing algorithms in a functional/categorical way is not only more elegant, concise and less error-prone, but provides numerous more tangible benefits, such as automatic parallelisation and distribution of algorithms.
Imperative pseudo-code

1: function $\text{MonteC}(n)$
2:  $x \leftarrow 0$
3:  for $i \leftarrow 1$ to $n$ do
4:      draw $u \sim U(0, 1)$
5:      draw $v \sim U(0, 1)$
6:      if $u^2 + v^2 < 1$ then
7:         $x \leftarrow x + 1$
8:  end if
9:  end for
10: return $4x/n$
11: end function

1: function $\text{Metrop}(n, \varepsilon)$
2:  $x \leftarrow 0$
3:  for $i \leftarrow 1$ to $n$ do
4:      draw $z \sim U(-\varepsilon, \varepsilon)$
5:      $x' \leftarrow x + z$
6:      $A \leftarrow \phi(x')/\phi(x)$
7:      draw $u \sim U(0, 1)$
8:      if $u < A$ then
9:         $x \leftarrow x'$
10:     end if
11: end for
12: return $x$
13: end function

Not obvious that one of these is naturally parallel...
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
• 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-kindred 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.
• A collection of type $M[T]$ can contain (multiple) values of type $T$

• If the collection supports a higher-order function $\text{map}(f: T \mapsto S): M[S]$ then we call the collection a **Functor**
  • eg. $\text{List}(1,3,5,7) \text{ map } (x \mapsto x*2) = \text{List}(2,6,10,14)$

• If the collection additionally supports a higher-order function $\text{flatMap}(f: T \mapsto M[S]): M[S]$ then we call the collection a **Monad**
  • eg. $\text{List}(1,3,5,7) \text{ flatMap } (x \mapsto \text{List}(x,x+1))$
    = $\text{List}(1, 2, 3, 4, 5, 6, 7, 8)$
  • instead of $\text{List}(1,3,5,7) \text{ map } (x \mapsto \text{List}(x,x+1))$
    = $\text{List(\text{List}(1,2),\text{List}(3,4),\text{List}(5,6),\text{List}(7,8))}$
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 \( f: S \rightarrow T, g: T \rightarrow U, h: U \rightarrow V \), we can compose them as \( h \circ g \circ f \) or \( s \rightarrow h(g(f(s))) \) 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 \( \text{flatMap} \):
  \[ s \rightarrow f(s) \text{ flatMap } g \text{ flatMap } h \] has type \( S \rightarrow M[V] \)

- Can be written as a for-comprehension (do in Haskell):
  \[ s \rightarrow \text{for } (t<-f(s); u<-g(t); v<-h(u)) \text{ yield } v \]

- Just syntactic sugar for the chained flatMaps above — really not an imperative-style “for loop” at all...
• 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.
• `Futures` can be transformed using `map` and `flatMap` irrespective of whether or not the `Future` computation has yet completed and actually contains a value.
• `Futures` are a powerful method for developing parallel and concurrent programs in a modular, composable way.
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**.
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).
  - \( \text{Vector}(3,5,7) \) map \((\_ \times 2)\) = \( \text{Vector}(6,10,14) \)
  - \( \text{Vector}(3,5,7).\text{par} \text{ map} (\_ \times 2) = \text{ParVector}(6,10,14) \)
- We can carry out reductions as folds over collections: \( \text{Vector}(6,10,14).\text{par} \text{ reduce} (\_ + \_) \) = 30
- In general, sequential folds cannot 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
- folds, scans 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), sometimes called foldMap
Log-likelihood calculation for iid model

Given a log–likelihood for a single observation, one can create a function to evaluate the full log-likelihood that is completely parallelisation–agnostic

```scala
def lli(th: Param, x: Obs): Double = ???

def ll(x: GenSeq[Obs])(th: Param): Double =
  x map (lli(th, _)) reduce (_+_)```

- If `ll` is initialised with a serial collection containing the observations, then the likelihood will be evaluated sequentially.
- If `ll` is initialised with a parallel collection, the likelihood will be evaluated in parallel on all available cores.
Distributed parallel collections with Apache Spark

- **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
- Good framework for the development of scalable algorithms for Bayesian computation
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 $f : x \rightarrow y$.

- Morphisms are closed under composition, so that if $f : x \rightarrow y$ and $g : y \rightarrow z$, then there must also exist a morphism $h : x \rightarrow z$ 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 $\text{id}_x : x \rightarrow x$, with the property that for any $f : x \rightarrow y$ 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 \( F : \mathcal{C} \to \mathcal{D} \) is a pair of functions (both denoted \( F \)):
  - \( F : \text{ob}(\mathcal{C}) \to \text{ob}(\mathcal{D}) \)
  - \( F : \text{hom}(\mathcal{C}) \to \text{hom}(\mathcal{D}) \), where \( \forall f \in \text{hom}(\mathcal{C}) \), we have \( F(f : x \to y) : F(x) \to F(y) \)
  - 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 \( F : \mathcal{C} \to \mathcal{C} \) is called an **endofunctor** — in the context of functional programming, the word functor usually refers to an endofunctor \( F : \text{Hask} \to \text{Hask} \)
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 $F, G : C \rightarrow D$
- A natural transformation $\alpha : F \Rightarrow G$ is a family of morphisms in $D$, where $\forall x \in C$, the component $\alpha_x : F(x) \rightarrow G(x)$ is a morphism in $D$
- To be considered natural, this family of morphisms must satisfy the naturality law:
  $$\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \rightarrow y \in \text{hom}(C)$$
- 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 $\mathbb{M}$ (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 $C$, the Kleisli category of $C$, written $C_T$ is a category with the same objects as $C$, but with morphisms given by:
  - $\text{hom}_{C_T}(x, y) = \text{hom}_C(x, T(y))$, $\forall x, y \in \text{ob}(C)$
- The identity morphisms in $C_T$ are given by $\text{id}_x = \eta(x), \forall x$, and morphisms $f : x \rightarrow T(y)$ and $g : y \rightarrow T(z)$ in $C$ 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 $C_T$.
- 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)
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 $W : \mathcal{C} \to \mathcal{C}$ together with two natural transformations $\varepsilon : W \to \text{Id}_\mathcal{C}$ (counit) and $\delta : W \to W^2$ (comultiplication) fulfilling the comonad laws:
  - **Associativity:** $\delta_W \circ \delta = W\delta \circ \delta$, as transformations $W \to W^3$
  - **Identity:** $\varepsilon_W \circ \delta = W\varepsilon \circ \delta = 1_W$, as transformations $W \to W$

- 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.
Laziness, composition, laws and optimisations

- Laziness allows some optimisations to be performed that would be difficult to automate otherwise
- Consider a dataset \( \text{rdd}: \text{RDD}[T] \), functions \( f: T \rightarrow U \), \( g: U \rightarrow V \), and a binary operation \( \text{op}: (V,V) \rightarrow V \) for monoidal type \( V \)
- We can map the two functions and then reduce with:
  - \( \text{rdd map } f \text{ map } g \text{ reduce } \text{op} \)
  - to get a value of type \( V \), all computed in parallel
- However, re-writing this as:
  - \( \text{rdd map } (g \text{ compose } f) \text{ reduce } \text{op} \)
  - would eliminate an intermediate collection, but is equivalent due to the 2nd functor law (map fusion)
- 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, Monad and Comonad

```scala
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
}
```
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 flatMap[A,B,D[B]] <: GenTraversable[B](ca: C[A])(f: A => D[B]): C[B]
  def reduce[A](ca: C[A])(f: (A, A) => A): A
  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 (serial) collections (eg. `Vector`), parallel collections (eg. `ParVector`), and distributed parallel collections (eg. `RDD`)

• We can then write code that is completely parallelisation-agnostic
Scalable ABC

- Simulate a collection of parameters, \( \text{pars: } C[\text{Param}] \), from a prior and store in a generic collection
- Define essential features of the ABC model:
  
  \[
  \begin{align*}
  \text{realData: } & \text{ Data} \\
  \text{simModel: } & \text{ Param } \Rightarrow \text{ Data} \\
  \text{summ: } & \text{ Data } \Rightarrow \text{ SS} \\
  \text{dist: } & (\text{SS, SS} ) \Rightarrow \text{ Double}
  \end{align*}
  \]

- Then scalable (parallelism–agnostic) ABC is as simple as:

  \[
  \text{val } \text{realSS = summ(realData)} \\
  \text{pars.map(summ compose simModel).} \\
  \text{filter(dist(_, realSS) < eps)}
  \]

  using map–fusion to avoid constructing a collection of synthetic datasets
A scalable particle filter

- We can write a scalable particle filter by first writing a generic (parallel–agnostic) function to update a cloud of particles using a single observation, and then apply this to a sequence of observations.

```scala
type PFState = GenericColl[State]

val update: (PFState, Obs) => PFState = ???

val data: Foldable[Obs] = ???
val pi0: PFState = ???

val piN = data.foldLeft(pi0)(update)
```
Composable time series models

- [git.io/statespace](https://git.io/statespace) (Student: Jonny Law)
- Scala library for nonlinear statespace modelling in continuous time for irregularly spaced observations on intractable Markov processes using particle filtering
- Compositional API for modelling building and use on-line with a streaming API based on Akka Streams

```scala
val sde = Sde.brownianMotion(1)
val negBinModel = Model.negativeBinomial(sde)
val sde2 = Sde.ouProcess(8)
val composedMod = negBinModel |+| Model.seasonal(24, 4, sde2)
```

Another library: [git.io/dlm](https://git.io/dlm) for more conventional linear statespace modelling
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.
Let’s start with writing a function to compute the weighted average of values at the start of an infinite data Stream.

```scala
def linFilter(weights: Stream[Double])(s: Stream[Double]): Double =
  (weights, s).parMapN(_*$_).sum
```

We can extend this local computation to the entire Stream using coflatMap.

```scala
s.coflatMap(linFilter(Stream(0.25, 0.5, 0.25)))
```

resulting in a new infinite Stream containing the filtered values.

- Note that the method extract will return the value at the head of the Stream.
Filtering an Image

- For a 2D Image, the relevant context for local computation is less clear, so to become comonadic, the Image class needs to be “pointed”, by augmenting it with a “cursor” pointing to the current pixel of interest
- For the pointed image class, `extract` returns the value of the pixel at the cursor
- We filter using functions which compute a value on the basis of the “neighbourhood” of the cursor
- `map` and `coflatMap` operations will automatically parallelise if the image is backed with a parallel data structure

```scala
def fil(pi: PImage[Double]): Double = (2*pi.extract + pi.up.extract + pi.down.extract + pi.left.extract + pi.right.extract) / 6.0
val filteredImage = pim0.coflatMap(fil)
def pims = Stream.iterate(pim0)(_ . coflatMap(fil))```
Gibbs sampling (an Ising model)

```scala
def gibbsKernel(pi: PImage[Int]): Int = {
  val sum = pi.up.extract + pi.down.extract +
             pi.left.extract + pi.right.extract
  val p1 = math.exp(beta * sum)
  val p2 = math.exp(-beta * sum)
  val probplus = p1 / (p1 + p2)
  if (new Bernoulli(probplus).draw) 1 else -1
}

def oddKernel(pi: PImage[Int]): Int =
  if ((pi.x + pi.y) % 2 != 0) pi.extract else gibbsKernel(pi)

def evenKernel(pi: PImage[Int]): Int =
  if ((pi.x + pi.y) % 2 == 0) pi.extract else gibbsKernel(pi)

def pims = Stream.iterate(pim0)(_.coflatMap(oddKernel).coflatMap(evenKernel))
```
Recursion schemes

- A **fold** over a sequential data structure is an example of a recursion scheme known more formally as a **catamorphism** (eg. sums and AD).
- The generation of a sequential stream of values from a seed and a generation function (eg. the generation of an MCMC chain) is an example of an **unfold** operation, more formally known as an **anamorphism**.
- Generating and then consuming a data structure (eg. simulating an MCMC chain and then computing summary statistics from the iterations) is an example of a **refold**, known formally as a **hylo**morphism.

(glossing over **F-algebras, coalgebras, ADTs and fixed point types**)

Again, category theory provides the theory and FP languages provide frameworks and abstractions for implementation.
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.
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
  \[
  \text{Exponential: } \text{Double} \rightarrow \text{Rand[Double]}
  \]
  and a Poisson parametrised by its mean as a function
  \[
  \text{Poisson: } \text{Double} \rightarrow \text{Rand[Int]}
  \]
- Those two functions don’t directly compose, but do in the Kleisli category of the **Rand** monad, so
  \[
  \text{Exponential}(3) \text{ flatMap } \{\text{Poisson(\_)}\} \]
  will return a \text{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.

```plaintext
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.
- Suppose we have a computer model, $f$, mapping inputs (or parameters) $x \in X$ to output $y \in Y$:

$$f : X \rightarrow Y$$

- Suppose further that we are uncertain about the “correct” inputs to use, and encapsulate our uncertainty in a probability distribution $p_X \in P(X)$, where $P$ denotes a probability monad.

- We can’t apply $f$ to $p_X$ (the types don’t match), but we can use the fact that $P$ is a functor to “lift” $f$ into the monad $(px \text{ map } f)$:

$$P(f) : P(X) \rightarrow P(Y),$$

inducing a probability distribution on the output.
Suppose now that we have a stochastic (non-deterministic) computer model (or emulator):

\[ g : X \rightarrow P(Y), \]

so that for a given fixed input \( x \in X \), the output is a probability distribution over \( Y \).

If we have additional uncertainty for the input \( x \in X \), then we lift \( g \) into the Kleisli category of the monad \( (px \, \text{flatMap} \, g) \):

\[ \mu_Y \circ P(g) : P(X) \rightarrow P(Y), \]

effectively just marginalising over our uncertainty about \( x \).
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 $x: \text{Rand}[X]$, monadic bind $x \text{ flatMap } (f: X \Rightarrow \text{Rand}[Y])$ returns $y: \text{Rand}[Y]$.
- $f$ 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.
Gen for property-based testing

- It turns out that monads representing non-determinism are widely used for software testing in FP languages.
- Property-based testing is an alternative to unit testing, and is concerned with specifying the properties that your function should satisfy rather than focusing on specific test cases.
- Having specified the properties, the testing library then randomly generates test cases for stress-testing functions to ensure correct behaviour.
- While not representing a true probability monad (distributional properties are unspecified), these generators have much in common with forward simulation probability monads.
- ScalaCheck in Scala and QuickCheck in Haskell are well-known examples of property-based testing libraries.
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 introduce 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)
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)
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
• 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, comonads, natural transformations and recursion schemes 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


For more about Scala and FP: darrenjw.wordpress.com