A compositional approach to statistical computing, machine learning, and probabilistic programming

Darren Wilkinson

@darrenjw darrenjw.github.io

Newcastle University, UK / The Alan Turing Institute

Bristol Data Science Seminar

Bristol University

5th February 2020

Overview

Background

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

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

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 MonteC(n)
2:     x ← 0
3:     for i ← 1 to n do
4:         draw u ~ U(0,1)
5:         draw v ~ U(0,1)
6:         if u^2 + v^2 < 1 then
7:             x ← x + 1
8:         end if
9:     end for
10:    return 4x/n
11: end function
```

```
1: function Metrop(n, ε)
2:     x ← 0
3:     for i ← 1 to n do
4:         draw z ~ U(−ε, ε)
5:         x′ ← x + z
6:         A ← φ(x′)/φ(x)
7:         draw u ~ U(0,1)
8:         if u < A then
9:             x ← x′
10:        end if
11:    end for
12:    return x
13: end function
```

Not obvious that one of these is naturally parallel. . .

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

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 =>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/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)
  - `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
- `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

```
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

Category theory

- A category $\mathcal{C}$ consists of a collection of objects, $\mathrm{ob}(\mathcal{C})$, and morphisms, $\mathrm{hom}(\mathcal{C})$. 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 $\mathrm{hom}_{\mathcal{C}}(x, y)$.
  $f \in \mathrm{hom}_{\mathcal{C}}(x, y)$ 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 $f, g, h \in \mathrm{hom}(\mathcal{C})$.
- 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

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):
  - $F : \mathrm{ob}(\mathcal{C}) \longrightarrow \mathrm{ob}(\mathcal{D})$
  - $F : \hom(\mathcal{C}) \longrightarrow \hom(\mathcal{D})$, where $\forall f \in \hom(\mathcal{C})$, we have $F(f : x \longrightarrow y) : F(x) \longrightarrow F(y)$
  - In other words, if $f \in \hom_{\mathcal{C}}(x, y)$, then $F(f) \in \hom_{\mathcal{D}}(F(x), F(y))$
- 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 $f, g \in \hom(\mathcal{C})$
- 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:
  - $\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \longrightarrow y \in \mathrm{hom}(\mathcal{C})$
- 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 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:
  - $\hom_{\mathcal{C}_T}(x, y) = \hom_{\mathcal{C}}(x, T(y)),\ \forall x, y \in \mathrm{ob}(\mathcal{C})$
- 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 $\mathcal{C}_T$.
- 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 $W : \mathcal{C} \longrightarrow \mathcal{C}$ together with two natural transformations $\varepsilon : W \longrightarrow \mathrm{Id}_{\mathcal{C}}$ (counit) and $\delta : W \longrightarrow W^2$ (comultiplication) fulfilling the comonad laws:
  - Associativity: $\delta_W \circ \delta = W\delta \circ \delta$, as transformations $W \longrightarrow W^3$
  - Identity: $\varepsilon_W \circ \delta = W\varepsilon \circ \delta = 1_W$, as transformations $W \longrightarrow W$
- 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 `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 (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

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.

Linear filters for Streams and Images

Let's start with writing a function to compute the weighted average of values at the start of an infinite data `Stream`

```
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`

```
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

```
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))
```

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)

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.
- Law, J., Wilkinson, D. J. (2019) Functional probabilistic programming for scalable Bayesian modelling, *arXiv*, 1908.02062.
- Park, S., Pfenning, F., Thrun, S. (2008) A probabilistic language based on sampling functions, *TOPLAS*, **31**(1):4.
- Ścibior, A., Kammar, O., Gharamani, Z. (2018) Functional programming for modular Bayesian inference, *Proc. ACM Prog. Lang.*, **2**(ICFP): 83.

Rainier: `rainier.fit`

`darrenjw.github.io` `@darrenjw`