A compositional approach to Bayesian computation

ISBA 2021 virtual world meeting

Darren Wilkinson

darrenjw.github.io

Newcastle University, UK

Introduction

Background

- For non-trivial problems, Bayesian computation typically relies on computationally intensive methods, such as MCMC, SMC or ABC
- These often require a custom implementation in some programming language
- All of the languages commonly used are very old, dating back to the dawn of the computing age, and quite unsuitable for scalable and efficient Bayesian computation
- Interpreted dynamic languages such as R, Python and Matlab are far too slow, among many other things
- Languages such as C, C++, Java and Fortran are much faster, but are also very poorly suited to the development of efficient, well-tested, scalable code, able to take advantage of modern computing hardware

Alternative languages and approaches

- All of the languages on the previous slide are fundamentally *imperative* programming languages, mimicking closely the way computer processors actually operate
- There have been huge advances in computing science in the decades since these languages were created, and many new, different and better programming languages have been created
- Although *functional* programming (FP) languages are not new, there has been a large resurgence of interest in functional languages in the last decade or two, as people have begun to appreciate the advantages of the functional approach, especially in the context of developing large, scalable software systems, and the ability to take advantage of modern computing hardware
- There has also been a swing away from *dynamically typed* programming languages back to *statically typed* languages

Compositionality and functional programming

Functional programming

- FP languages emphasise the use of *immutable* data, *pure, referentially transparent functions*, and *higher order functions*
- FP languages more naturally support composition of models, data and computation than imperative languages, leading to more scalable and testable software
- Statically typed FP languages (such as Haskell and Scala) correspond closely to the *simply-typed lambda calculus* which is one of the canonical examples of a *Cartesian closed category* (CCC)
- This connection between typed FP languages and CCCs enables the borrowing of ideas from category theory into FP
- Category theory concepts such as *functors*, *monads* and *comonads* are useful for simplifying code that would otherwise be somewhat cumbersome to express in pure FP languages

Concurrency, parallelism, distribution, state

- Modern computing platforms feature processors with many cores, and possibly many such processors - parallel programming is required to properly exploit this
- Most of the notorious difficulties associated with parallel programming revolve around *shared mutable state*
- In pure FP, state is not mutable, so there is no mutable state, shared or otherwise
- Consequently, most of the difficulties typically associated with parallel and concurrent programming simply don't exist in FP - parallelism in FP is so easy and natural that it is sometimes completely automatic
- This natural scalability of FP languages is one reason for their recent resurgence

Compositionality

- Not all issues relating to scalability of models and algorithms relate to parallelism
- A good way to build a large model is to construct it from smaller models
- A good way to develop a complex computation is to construct it from simpler computations
- This (recursive) decomposition-composition approach is at the heart of the so-called "divide and conquer" approach to problem solution, and is very natural in FP
- It also makes code much easier to *test* for correct behaviour
- Category theory is in many ways the mathematical study of (associative) composition, and this leads to useful insights

Bayesian computation

- *map-reduce* operations on *functorial* data collections can trivially parallelise (and distribute):
  - Likelihood evaluations for big data
  - ABC algorithms
  - SMC re-weighting and re-sampling
- Gibbs sampling algorithms can be implemented as *cobind* operations on an appropriately coloured (parallel) *comonadic* conditional independence graph
- *Probabilistic programming languages* (PPLs) can be implemented as embedded domain specific languages (DSLs) trivially using *for/do* syntax for *monadic composition* in conjunction with *probability monads*
- Automatic differentiation (AD) for compute graphs is particularly natural in functional languages, facilitating gradient-based algorithms such as MALA, HMC and PDMP

Conclusion

Summary

- All of the programming languages commonly used for Bayesian computation are poorly suited to the task
- Functional programming languages borrowing ideas from *Cartesian closed categories* provide an excellent foundation for scalable modelling and computation
  - Concepts from category theory, such as *functors*, *monads* and *comonads* provide a useful framework for organising computation
- Functional languages are particularly elegant and powerful for probabilistic and differentiable programming
- This isn't about one particular programming language (eg. *Scala* or *Haskell*), or language syntax - it is about a class of languages:
  - Statically typed compiled functional programming languages with type inference and higher-kinded types

Further information

github.com/darrenjw/isba2021