Functional Programming through Deep Time: Modeling the first complex ecosystems on Earth

Experience Report

Emily G. Mitchell
University of Cambridge & British Antarctic Survey, Cambridge, UK
ek338@cam.ac.uk

Abstract

The ecology of Earth's first large organisms is an unsolved problem in palaeontology. This experience report discusses the determination of which ecosystems could have been feasible, by considering the biological feedbacks within them. Haskell was used to model the ecosystems for these first large organisms – the Ediacara biota. For verification of the results, the statistical language R was used. Neither Haskell nor R would have been sufficient for this work – Haskell's libraries for statistics are weak, while R lacks the structure for expressing algorithms in a maintainable manner. This work is the first to quantify all feedback loops in an ecosystem, and has generated considerable interest from both the ecological and palaeontological communities.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages, Experimentation

Keywords Haskell, R, Palaeontology, Ecology

1. Introduction

Complex life on Earth evolved 600 million years ago, after billions of years of simple microbial life. The first complex organisms were the Ediacara biota, which lasted only 30 million years – a blink of a geological eye. The Ediacara biota were shortly followed by the Cambrian Explosion, bringing with it the precursors to modern life, which have dominated the Earth ever since. Understanding why these Ediacara biota failed can give clues to how ecosystems function. These unsuccessful organisms are unlike anything else, so many traditional techniques from biology and palaeontology do not apply. Computer modelling can give us new insights by allowing us to test theories, including some that have been debated for over 40 years.

Rangeomorphs are a group of Ediacaran species, with a fractal branching structure, which maximizes surface area – see Figure 1. Organisms that maximize their surface area normally feed in one of three ways:

1. *Photosynthetic* - converting sunlight to energy [8].
2. *Suspension feeding* - filtering plankton from the water column [6].
3. *Osmotrophic* - absorbing organic carbon directly through their membrane walls [7].

We now know that most rangeomorphs lived in the deep ocean, so can't have been photosynthetic [15], but the debate rages on between the other two strategies. Using Haskell, I modelled potential Ediacaran ecosystem as graphs, with species as nodes and feeding relationships as edges [10]. From these graphs the feeding strategies corresponding to feasible ecosystems were determined. Most rangeomorphs were found to be osmotrophic.

How fossils are spatially distributed in the rock gives clues to their interactions in life. These distributions were used to validate my models by comparing feasible ecosystems to those suggested by the actual fossils. To extract a graph from spatial positions I used two approaches. Firstly, the programming language R [11] was used to compare the actual locations of fossils against a random layout (generated using Poisson processes). Monte Carlo simulation was used to quantify the significance of any variation. Secondly, Bayesian network inference was used on the spatial data to search for the most probable graph. To perform Bayesian network inference, a Haskell script was used which invokes Banjo, a program written in Java [13].

This experience report first discusses the use of computer programs in ecology and palaeontology in §2. The work is described in §3, then, since R is both the most commonly used language in ecology and the language this project was started in, R is compared to Haskell in §4. §5 describes how this work has been received by the wider ecological and palaeontological communities, and gives advice to colleagues choosing a programming language.

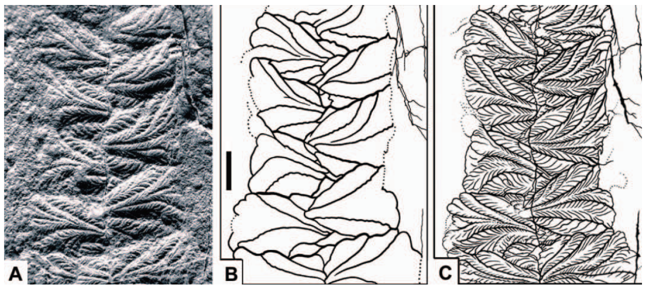

Figure 1. *Fractofusus misrai*, a type of Rangeomorph. A is a photo of the original fossil, B shows the primary branches and C shows the primary and secondary branches. The scale bar is 1cm. Taken from [2]

2. Language use in Ecology and Palaeontology

Ecology and palaeontology are both diverse subjects, ranging from qualitative descriptive work to theoretical work. Computer literacy varies greatly in both fields, from people who don't use computers even for word processing, to former computer scientists. Out of the two groups, ecologists tend to be more computer literate.

2.1 Ecologists

Computational and theoretical ecologists have the highest knowledge of computer science. These two areas require complex algorithms, and languages such as C/C++, Fortran and Perl are used. Ecology often requires statistical analysis, so most ecologists have experience using either a specialized GUI, or a programming language, typically R [11]. There are a number of reasons why R is preferred for statistics:

- There are many workshops and courses in R, from basic introductions to applying the latest statistical methods.
- Books on statistical ecology often give examples in R, and have R scripts available for download [5].
- Statisticians and computational ecologists often develop new techniques in R, resulting in a large number of specialist packages.
- Papers often reference R packages, documenting their applications and limitations.

2.2 Palaeontology

Palaeontology is often qualitative, so it is unusual for palaeontologists to have experience with programming languages, or even command line driven programs. Both quantitative and computational palaeontology are dominated by GUIs specifically written for palaeontologists [3] (apart from the small field of macroevolutionary modelling – which is dominated by C++). Additionally, Microsoft Excel is used for plots and regressions. As palaeontology starts to become more quantitative, there is a small but growing trend towards using R, with palaeontological R courses starting to be taught at masters level and above.

2.3 Choosing Haskell

Prior to entering the field of palaeontology, most of my previous research was in analytical mathematical physics. My main programming experience was in R, and I had also used some Matlab and C++. For my work in palaeontology, I initially used R.

I started considering alternative languages when I had trouble expressing more complex algorithms in R. I was aware of Haskell as my husband both works as a functional programmer and is active in the Haskell community. One motivation for choosing Haskell was to take advantage of his expertise. However, I found our programming styles surprisingly incompatible, and in practice programmed almost entirely independently.

My husband puts great emphasis on code that is "beautiful" – that is, economical, reusable, modular. This focus seems widespread in the Haskell community. In contrast, I am happy with code that works – once I have written a function, I rarely revisit it. I suspect that my style is widespread in scientific programming.

My primary resource when learning Haskell was Hutton [4], which I found to be clear, and had many suitable exercises. I read a variety of online Haskell tutorials (including Learn You a Haskell for Great Good), but did not find these particularly helpful – none had exercises which gradually increased in difficulty.

I wrote my Haskell code in TextPad on Windows XP, experimented on it using GHCi and compiled it using GHC [14]. I made light use of the built in GHC profiler. Once I had reached a basic level in my understanding of types, Hoogle [9] was useful for finding functions.

3. Haskell for Computational Palaeoecology

My work involves using biological interactions to understand the development of ancient ecosystems [1]. An ecosystem can be modelled as a graph, where species are represented as nodes, and interactions as edges. Feedback mechanisms are one way of analysing the structure of ecosystems, and these are represented by loops in the graph [10]. I first considered the relationship between feedback loops and stability for modern ecosystems, then applied the insights gained to Ediacaran ecosystems.

3.1 Technique Development

Properties of feedback loops have been shown to indicate ecosystem stability [10]. Previous work focused on "biologically interesting" feedback mechanisms, which account for less than 1% of the total number of loops. The natural extension to this work is to consider every type of loop. There is a R package which qualitatively finds all the loops in a graph (`LoopAnalyst`), however this package would have required significant adapting to be useful for my analysis. Therefore, I implemented a program in R that was able to quantitatively analyse all loops, and output the most significant.

Analysis of all loops revealed an unexpected result regarding ecosystem stability. To confirm this finding, variations on how the loops were computed and analysed needed to be explored. This exploration was problematic in R for two reasons, the program didn't run fast enough, and program modification was fiddly. Improvement of the algorithm in R would have made it more intricate, and thus even harder to modify. Writing a second version of the program in Haskell solved both these problems. A more advanced algorithm was used, but thanks to static typing and algebraic data types, modification was straightforward – the compiler did lots of the hard work. With a significantly improved algorithm, and with the advanced compilation techniques of GHC, the program ran over 10,000 times faster.

Using the Haskell program, possible relationships between loops and stability were investigated, which allowed confirmation of my initial results. Given my programming level, I would not have been able to get the same results using R, Matlab or C++. For this work, Haskell provided an easily modifiable language that ran quickly – I can't think of any improvements to Haskell which would have made it more suitable.

3.2 Application to Palaeontology

The Haskell program was used to investigate the plausibility of different feeding strategies of Ediacaran species. For each permutation of feeding strategy, an ecosystem graph was created, all loops found, and stability calculated. Using these stabilities, I showed that most of the early Ediacaran species must have been osmotrophic.

During this modelling work, the Haskell program was run hundreds of thousands of times, which would not have been feasible with the R program. For this type of work, seconds quickly add up, so a faster program is always better – but the Haskell program was fast enough. Finding Haskell packages to produce the right type of plot was hard, so the data was saved and viewed in R.

It is important to verify any conclusions drawn from modelling using statistical analysis of real data. I collected data from fossil sites in order to be able to statistically analyse the distributions of fossils on the rock face, allowing a network of fossil interactions to be built up. The book I used to learn the appropriate statistical techniques included examples in R, making it easy to use R for my analysis. I looked at the statistical packages available for Haskell, but the techniques I required were not supported (see §4.4). Switching back to R was the obvious choice.

4. Comparing Haskell and R

My work has used both Haskell and R, so this section is a comparison between them. R is a language designed for "statistical computing and graphics" [11]. R defers evaluation of function arguments, but is otherwise a strict language, R is impure and has no static typing; it has some functional aspects, but is typically used in an imperative manner. In contrast, Haskell is a lazy, pure, statically-typed functional language.

Writing simple programs in R is easy, but writing more complex programs can get messy. In contrast, writing simple programs in Haskell requires a greater understanding of the language (such as monads, types, etc.), but producing complicated programs is not much harder.

4.1 Syntax and Reasoning

I prefer the Haskell syntax because it is consistent, and has an obvious correspondence with mathematical concepts. This correspondence means that colleagues with no Haskell or functional programming experience can clearly understand my calculations. In comparison, the syntax for R is more verbose, and less consistent, but some statistical functions are written in a intuitive way – for example regressions are written `lm(`$z \sim x + y$`)`. Unfortunately, these syntactic shortcuts can be confusing in themselves, for example polynomial regressions are then written `lm(`$z \sim I(x^3) + y$`)`. The consistent syntax of Haskell means that when returning to Haskell after a break, getting back up to speed is quicker compared to returning to R.

Haskell's purity allows extensive use of equational reasoning – for example the code `if a then b else b` is equivalent to `b` (providing `a` terminates). In contrast, the equivalent R transformation is only true if `a` both terminates and does not produce side effects, which can be subtle, such as coercion changing a variable's underlying type. I have made such mistakes in R, which means it is harder to refactor my R code, and thus my Haskell code is cleaner. Similarly, when debugging Haskell I am able to deduce properties about my code using only local knowledge, whereas R requires a much more advanced understanding of both the program and the language.

The downside of purity is the requirement to use monadic IO. I found the descriptions of IO in tutorials obtuse, but found the reality to be significantly simpler. The only awkwardness is that switching between pure and monadic code feels like changing mode, the syntax is different and so are many of the functions (e.g. `map` vs. `mapM`).

4.2 Types

The static type system of Haskell is difficult to adjust to at first, but worth the struggle – the reduction in debugging time is significant. R has no static types, and makes significant use of coercions. If you try and use a string as a matrix, the string will be parsed as a matrix, which is particularly useful when reading data tables from a file. If you try and use a vector as a matrix, a matrix will be created by repeating the vector.

These coercions make debugging hard, and significantly reduce the confidence in numerical results. Many bugs that result in compile time type errors in Haskell, result in wrong (but plausible) results in R. Additionally, coercions can be handled somewhat inconsistently, e.g. a matrix produced by converting a list using `as.matrix` will cause a runtime type error when used with some matrix functions.

Static types are particularly helpful for beginners to find basic library functions, using Hoogle. For example, the type of the function to remove duplicates from a list is obvious (`Eq a => [a] -> [a]`), but it's name in Haskell is obscure (`nub`).

Sadly, Haskell packages do not always use consistent types – for example there are many different vectors/arrays, which makes using multiple packages painful. Some R packages also have different types, but these problems are entirely masked by coercions, meaning lots of R packages can be used together with ease.

4.3 Complex Code

Much of my work involves calling library functions, combining the results, and doing calculations which are mathematically taxing, but can be expressed fairly directly in any language. However, some of my work does require more careful thought about algorithms and more intricate structural manipulations. I find it significantly easier to write this complex code in Haskell, for three reasons:

Static types have an enormous impact on reliability, but for more complex code, they also provide a clear pattern for combining functions. Static types allow chunks of code to be combined like a jigsaw puzzle.

Algebraic data types allow me to accurately encode my data structures, and combined with the static type checking, make managing complexity much easier.

Higher-order functions allow detangling complex code, splitting out specific details from an algorithm. Both Haskell and R allow higher-order functions, but the lack of static type checking in R makes them much harder to use. In Haskell, using higher-order functions with algebraic data types, I was able to parameterize operations by the details I needed to tweak, making modifications straightforward.

4.4 Libraries

For the libraries I require, R has far better coverage than Haskell. The default installation of R has all the tools necessary for pre-university maths or statistics. In comparison, the standard Haskell installation (the Haskell Platform) includes advanced multithreading, but lacks a function to compute the mean. From the base installation, R can install new packages through the GUI, while Haskell requires the use of a command line tool, and far more background computer knowledge.

R has a huge range of specialist statistics packages available, including many cutting edge statistical techniques. In comparison, Haskell has nine statistics libraries, which have a reasonable amount of overlap, but skip things as basic as t-tests. One of the most important aspects of performing statistical tests is checking all the necessary assumptions are valid, something often overlooked in biological sciences [12]. Haskell packages provide normal distribution functions, but lack the Shapiro-Wilks Test which checks data for normality – a key assumption in most parametric tests.

R has a built in plotting environment, which easily produces a wide range of plots. Complex plots are provided in libraries, and most statistical packages integrate plotting functionality. In comparison, Haskell has a choice of many plotting libraries, all of which are significantly different, and none of which have the integration of R. In R a plot is only a simple function call away, and they are used continuously. In Haskell, a plot is significantly harder – I just export my data to R and plot it there.

As an example of the library issues discussed, the following R code plots the eigenvalues of a matrix read from disk:

```
plot(eigen(read.delim("matrix.txt"))$ values)
```

In Haskell finding eigenvalues involves installing `hmatrix` (which on Windows requires changing environment variables and setting Dll paths), reading a matrix requires parsing code, and plotting is never as simple as just calling `plot`. Performing this operation from a base install of R takes seconds, in Haskell it takes far more effort.

5. Evaluation

This section discusses how my work was received by the ecological and palaeontological communities, along with plans for my program and future use of Haskell. I conclude with advice on using Haskell for computational palaeoecology.

5.1 Reception

Ecologists are excited about my work. The work done by Neutel et al. [10] provoked much interest in loop analysis of ecological graphs, and because my program is quick and easy to use, people can now explore the importance of loops in much larger real and theoretical ecosystems.

My code is available on Hackage as `loopy`[1], but for my colleagues I provide a precompiled executable with an R wrapper script, which suits them well.

Palaeontologists are interested in my results, but not in my methods. Modelling ancient ecosystems as graphs is a new technique, and has yet to become mainstream. If I wanted palaeontologists to use my program, a GUI would probably be necessary, something I currently have no plans for.

5.2 Future Development

For huge graphs, it is infeasible to enumerate all loops (the number of loops can grow exponentially). I plan to explore approximations to allow larger graphs to be processed, which permit ecosystems to be modelled in finer detail. For this work, I will continue using Haskell, with R for any plotting or statistical analysis.

Haskell is my programming language of choice, and if it had adequate statistics libraries, I would rarely use anything else. If I ever need to run intensive statistical analysis on lots of large data sets, it would be worth writing the necessary libraries in Haskell.

Another language I am considering is F#. When I started this project, F# was a relatively new language, but now I think it could suit me quite well. F# provides access to the .NET libraries, which would provide much of the functionality missing in Haskell, while still keeping the benefits of functional programming and static typing.

5.3 Expansion of Haskell use in Ecology/Palaeoecology

Ecologists tend to use programming languages they have been taught at university, or languages already established in their research group. Once a language becomes dominant, it takes a long time to become unseated – Fortran is still being taught. Theoretical ecologists could find Haskell useful, because it's quicker to write complex code. However, the lack of simple integrated plots would be a severe obstacle. For statistical use, Haskell is unlikely to compete with R, which is deeply embedded in the community.

Computational palaeoecology is a new field – I am only aware of two other practitioners. Haskell is well suited for this domain, and if it became established, it would be unlikely for another language to take over. Whether it becomes established probably has more to do with the future career paths of existing computational palaeoecologists than with the language itself.

It would be difficult for Haskell to become the de facto language for scientific computing, due to its steep learning curve. Most undergraduate science degrees do not teach a separate programming course, instead teaching programming alongside a technique, such as statistics. Programming skills are gently built up in little steps, over several courses. Haskell could not be taught in this way, since static types and monads would need to be taught before getting to any technique, and this would probably be too much for one course.

Numerate disciplines like physics do have separate programming courses, where the choice of language depends on its usefulness post degree. If Haskell, or another functional language, became widely used commercially, universities would probably start to teach it outside the computer science department. These numerate disciplines could then write appropriate scientific libraries which could be used by less numerate sciences.

[1] `http://hackage.haskell.org/package/loopy`

5.4 Conclusions

When colleagues ask which language they should use, I only recommend Haskell if they are an experienced programmer, and deal mostly with theoretical work. For everyone else, I recommend R. While I think Haskell is a superior language, it has a much steeper learning curve, and unless I was willing to teach them, R is a simpler choice. Within my department there is an R learning group, and various university run courses, which give useful support.

Haskell enabled me to get detailed results that I could not otherwise have found. I find functional programming and static typing powerful concepts, and other than R, would not consider using languages lacking these features.

Acknowledgments

Neil Mitchell for introducing me to Haskell, Mike Dodds for feedback on this paper and my supervisors Nicholas Butterfield and Anje Neutel. This work was funded by a NERC studentship.

References

[1] N. J. Butterfield. Macroevolution and macroecology through deep time. *Paleontology*, 50:41–55, 2007.

[2] J. Gehling and G. Narbonne. Spindle-shaped Ediacara fossils from the Mistaken Point assemblage, Avalon Zone, Newfoundland. *Canadian Journal of Earth Sciences*, 44(3):367–387, 2007.

[3] Ø. Hammer, D. Harper, and P. Ryan. PAST: Paleontological statistics software package for education and data analysis. *Palaeontologia Electronica*, 4(1), 2001.

[4] G. Hutton. *Programming in Haskell*. CUP, 2007.

[5] J. Illian. *Statistical analysis and modelling of spatial point patterns*.

[6] R. Jenkins and J. Gehling. *A review of the frond-like fossils of the Ediacara assemblage*. South Australian Museum, 1978.

[7] M. Laflamme, S. Xiao, and M. Kowalewski. Osmotrophy in modular Ediacara organisms. *Proceedings of the National Academy of Sciences*, 106(34):14438, 2009.

[8] M. McMenamin. The garden of Ediacara. *Palaios*, pages 178–182, 1986.

[9] N. Mitchell. Hoogle overview. *The Monad.Reader*, (12):27–35, November 2008.

[10] A. M. Neutel, J. A. P. Heesterbeek, J. van de Koppel, G. Hoenderboom, A. Vos, C. Kaldeway, F. Berendse, and P. C. de Ruiter. Reconciling complexity with stability in naturally assembling foodwebs. *Nature*, 449:559–602, 2007.

[11] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[12] T. Siegfried. Odds are, it's wrong: Science fails to face the shortcomings of statistics. *Science News*, 177(7):26–29, 2010.

[13] V. Smith, J. Yu, T. Smulders, A. Hartemink, and E. Jarvis. Computational inference of neural information flow networks. *PLoS Computational Biology*, 2, 2006.

[14] The GHC Team. The GHC compiler, version 6.12.3. `http://www.haskell.org/ghc/`, 2010.

[15] D. Wood, R. Dalrymple, G. Narbonne, J. Gehling, and M. Clapham. Paleoenvironmental analysis of the late Neoproterozoic Mistaken Point and Trepassey formations, southeastern Newfoundland. *Canadian Journal of Earth Sciences*, 40(10):1375–1391, 2003.