

Experience Report: Functional Programming through Deep Time

Modeling the first complex ecosystems on Earth

Emily G. Mitchell

University of Cambridge
ek338@cam.ac.uk

Abstract

The ecology of Earth's first organisms is an unresolved problem in paleontology. I determine which ecosystems could have been feasible by considering the biological feedbacks within them. In my work, I use Haskell to model the ecosystems from the Ediacaran Biota – the first complex organisms. For verification of my results, I used the statistical language R. Neither Haskell nor R would have been sufficient for my work – Haskell's libraries for statistics are weak, while R lacks the structure for expressing algorithms in a maintainable manner. My work is the first to allow modeling of all feedback loops in an ecosystem, and has generated considerable interest from both the ecological and paleontological communities.

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

General Terms Languages, Experimentation

Keywords Haskell, R, Paleontology, Ecology

1. Introduction

Complex life evolved 600 million years ago, after billions of years of simple microbial life. The first complex organisms were the Ediacaran Biota, which lasted only 30 million years – a blink of a geological eye. These Ediacaran Biota were shortly followed by the Cambrian Explosion, bringing with it the precursors to modern life, which have dominated the world ever since. Understanding why these Ediacaran Biota failed can give clues to how ecosystems function. These unsuccessful organisms are unlike anything else, so many traditional techniques from paleontology and biology do not apply. Computer modeling 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 [7].

2. *Suspension feeding* - filtering out plankton from the water column [5].

3. *Osmotrophic* - absorbing organic carbon directly through their membrane walls [6].

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

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

In this experience report, I first discuss the use of computer programs in Ecology and Paleontology in §2. I describe my work in §3, then, since R is both the most commonly used language in Ecology and the language I used to start this project, I compare R to Haskell in §4. In §5 I describe how my work has been received by the wider ecology and paleontology communities, and give advice to colleagues choosing a programming language.

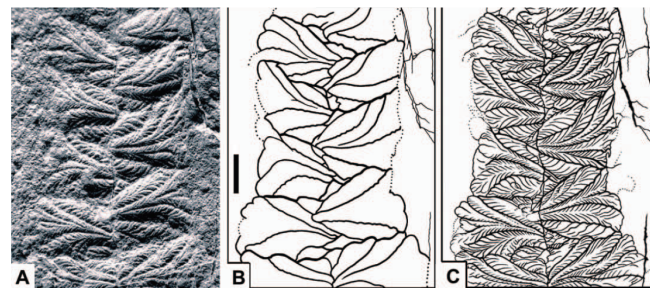


Figure 1. *Fractofusus*, a type of Rangeomorph. A is a photo of the original fossil, B shows the primary branches and C shows the primary and secondary branches. The scale bar is 1cm. Taken from [1]

2. What Paleontologists and Ecologists Use

Paleontology and ecology are very 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 downloadable R code [4].
- 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 Paleontology

Paleontology is often qualitative, so it is unusual for paleontologists to have experience with programming languages, or even command line driven programs. Both quantitative and computational paleontology are dominated by GUIs specifically written for paleontologists [2] (apart from the small field of macroevolutionary modeling – which uses C++). Additionally, Microsoft Excel is used for plots and regressions. As paleontology starts to become more quantitative, there is a small but growing trend towards using R, with some masters courses beginning to teach it.

2.3 Choosing Haskell

Prior to entering the field of paleontology, 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 paleontology, I initially used R.

I started considering alternative languages when I had trouble expressing more complex algorithms in R. My computer scientist husband suggested I learn Haskell, a language he uses daily. There was no readily available help for other programming languages, and the mathematical basis of Haskell appealed to me, so I started to learn Haskell.

My primary resource when learning Haskell was Hutton [3], 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, etc.), but did not find these particularly helpful – none had exercises which gradually increased in difficulty.

Initially I asked my husband for programming help, but his usual response was to suggest first refactoring my code – aiming to make it more “beautiful” – before considering the problem at hand. While Haskell is an elegant language, I just need code that works, so this fundamental difference in approach meant it was easier to work entirely independently¹. My husband was still useful for suggesting helpful packages.

¹ And also led to the creation of HLint, a tool to criticize Haskell code without personal interaction [8].

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, I suddenly found Hooghe [9] useful for finding functions.

3. Haskell for Computational Paleoecology

My work involves using biological interactions to understand the development of ancient ecosystems. Ecosystems can be modeled as a graph, where species are represented as nodes, and interactions as edges. Feedback mechanisms are one way of analyzing the structure of ecosystems, and these are represented by loops in the graph. 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 significant” feedback mechanisms, which accounted for less than 1% of the total number of loops. The natural extension to this work is to consider all loops, and attempts have been made using both Matlab and Perl – neither was able to produce all the loops. I implemented a program using R that was able to analyze all loops, and output the most significant.

Analysis of all loops revealed an unexpected indicator of ecosystem stability. To confirm this finding, I needed to explore many variations on how the loops were computed and analyzed. The initial R program had two problems, firstly it didn't run fast enough, and secondly program modification was fiddly. I could have improved the algorithm in R, but that would have made it more intricate, and thus even harder to modify. I wrote a second version of the program in Haskell, which solved both these problems. I used a more advanced algorithm, but thanks to static typing and algebraic data types, I was still able to modify it with confidence – the compiler did lots of the hard work for me. With a significantly improved algorithm, and with the advanced compilation techniques of GHC, the program ran over 10,000 times faster!

Using my Haskell program, I investigated possible relationships between loops and stability, which allowed me to confirm my initial results. 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 way Haskell could have been more suitable.

3.2 Haskell applied to Paleontology

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

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

It is important to verify any conclusions drawn from modeling using statistical analysis of real data. When learning the statistical techniques, the first book I read 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 in this section I compare them. R is a language designed for “statistical computing and graphics” [11]. R is strict, 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 no harder. Similarly, someone with limited program experience can easily start using R, but Haskell has a higher barrier to entry.

4.1 Syntax and Reasoning

I prefer the Haskell syntax because it is consistent, and has an obvious correspondence with mathematical notation. Haskell is so easy to read **TODO: delete?** 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 $\text{lm}(z \sim x + y)$. Unfortunately, these syntactic shortcuts can be confusing in themselves, for example polynomial regressions are then written $\text{lm}(z \sim I(x^3) + y)$. 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, including subtle side effects such as coercion changing a variables underlying type. I have made such mistakes in R, which means it is far 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, lots of Haskell packages have different types – for example there are many different vectors/arrays, which makes using multiple pack-

ages 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 that 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 make programming more like solving 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. For my work, using higher-order functions, I was able to parameterize operations by the details I needed to tweak – whereas with R the modifications were much harder to make. R does have higher-order functions, but they are not commonly used, and the lack of static type checking makes them much harder to get right.

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 install, 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, and skip things as basic as T-tests. One of the most important things about 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 for normality.

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, while in Haskell takes far longer, and would only be done if it fitted into a larger Haskell program.

5. Evaluation

In this section I discuss how my program was received by the ecological and paleontological communities, along with future directions for my program and my use of Haskell. I conclude with advice on using Haskell for computational paleoecology.

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, but the lack of a suitable tool made it much harder for anyone to continue this research. My program is quick and easy to use, so people can now explore the importance of loops in both real and theoretical ecosystems.

Most of the interest in my program has come from theoretical ecologists, who are comfortable with command line programs. GHC allows the creation of standalone executables, which can be easily used by any interested party. In comparison, converting an R program to an executable file is tricky – I found two packages which attempt this, neither worked for me.

Other ecologists who program R have also expressed an interest in using my program. For these people, I have wrapped my executable with an R script, which simply invokes my program, converting arguments given to the R function into command line arguments. This wrapping is simple, but effective. The only disadvantage is that R programmers would find it hard to modify my code.

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

5.2 Future Development

My program is currently limited to smaller networks. For large networks, the number of loops is infeasible to enumerate, so other approaches need to be explored. For example, instead of generating all loops, I could have an option to only generate loops with specific properties. Another potential option is finding loops at random, allowing properties to be estimated. These improvements would increase the applicability of my program, allowing ecosystems to be modeled in finer detail.

In addition to developing the program, there are a lot of ancillary resources that would help existing users, and attract new ones. The documentation is currently limited, and there is no tutorial material.

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 many of the libraries missing in Haskell.

5.3 Expansion of Haskell use in Ecology/Paleoecology

Ecologists tend to stick to 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 in regular use. Theoretical ecologists could find Haskell useful, because it's quicker to write complex code. However, the lack of easy and integrated plots would be a severe obstacle. For statistical use, Haskell is unlikely to compete with R, which is deeply embedded in the community. I think there would be some benefits to greater Haskell use, but I think it is unlikely to happen.

Computational paleoecology 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 paleoecologists than with the language itself.

5.4 My Advice

Haskell is now my programming language of choice, but if the task calls for lots of statistical analysis or plotting, I switch to R. If subsequently I need to run intensive statistical analysis on lots of large data sets, I would implement the required methods in Haskell, and then probably start to switch my statistics work to Haskell.

When colleagues ask which language they should use, I recommend Haskell only if they are an experienced programmer, and they 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 results that I could not otherwise have found. I am totally sold on the ideas of functional programming and static typing, and other than R, would not consider using languages lacking these features.

Acknowledgments

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

References

- [1] 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.
- [2] Ø. Hammer, D. Harper, and P. Ryan. PAST: Paleontological statistics software package for education and data analysis. *Palaeontologia Electronica*4. 2001.
- [3] G. Hutton. *Programming in Haskell*. Cambridge Univ Pr, 2007.
- [4] J. Illian. *Statistical analysis and modelling of spatial point patterns*.
- [5] R. Jenkins and J. Gehling. *A review of the frond-like fossils of the Ediacara assemblage*. South Australian Museum, 1978.
- [6] M. Laffamme, S. Xiao, and M. Kowalewski. Osmotrophy in modular Ediacara organisms. *Proceedings of the National Academy of Sciences*, 106(34):14438, 2009.
- [7] M. McMenamin. The garden of Ediacara. *Palaos*, pages 178–182, 1986. ISSN 0883-1351.
- [8] N. Mitchell. *HLint Manual*. URL <http://community.haskell.org/ndm/darcs/hlint/hlint.htm>.
- [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. Kaldewey, 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 6.8.2. <http://www.haskell.org/ghc/>, Dec. 2007.
- [15] D. Wood, R. Dalrymple, G. Narbonne, J. Gehling, and M. Clapham. Paleoenvironmental analysis of the late Neoproterozoic Mistaken Point and Trepassy formations, southeastern Newfoundland. *Canadian Journal of Earth Sciences*, 40(10):1375–1391, 2003.