Greg Michaelson

Introduction to Functional Programming through Lambda Calculus

Preface to Dover Edition, 2011

*Context*

When I started to write this book in 1986, functional programming seemed on an upward trajectory, out of academia into “real-world” computing. Spurred by the Japanese 5th Generation programme, many other nations initiated research and development schemes around “stateless” declarative programming languages. The Japanese projects laid great stress on hardware and software to support logic programming. In contrast, the UK Alvey programme gave equal emphasis to functional languages. Thus, major multi-University and industry projects sought to develop viable functional computing platforms at all levels, from VLSI and multi-processor hardware for graph reduction and combinators, like Coweb, Alice and GRIP, through novel programming languages, like Hope, Standard ML and Haskell, to fully integrated systems like the Flagship.

While there were many lasting research successes, overall the 5th Generation programmes utterly failed to make any impact on the von Neumann/imperative language status quo, and are now largely forgotten. In particular, declarative languages were oversold as the solution to the “software crisis”. The core claim was that the absence of state provided a model that greatly simplified high levels of abstraction, component based software construction, parallel implementation and proof of correctness. My original Introduction to this book is one such example of zeitgeist enthusiasm.

While declarative languages have substantial advantages for problems which are highly compositional in their data, the lack of state is a major hindrance for most substantive applications which do things *in* rather than *beyond* time. Similarly, while proving properties of declarative programs may, for small exemplars, be somewhat easier than for imperative code, this again depends on highly compositional program structure. The bottom line is that constructing and proving correct programs is hard, be it in a declarative or imperative language.

Nonetheless, research in functional programming has remained buoyant and its industrial significance is steadily growing, if somewhat more slowly than anticipated 25 years ago. For example, constructs from functional languages have found their way into mainstream imperative languages. Thus, functional abstraction is supported in Microsoft’s system language C#, the general purpose Ruby, the scripting language Python and the Java-derived Groovy. Furthermore, Microsoft Research supports the development of the Haskell language and has integrated its own Standard ML-derivedF# into the core .NET framework. Finally, cheap high performance computing on tens of thousands of commodity platforms has enabled international Internet businesses to build services on standardised parallel frameworks, based explicitly on higher order constructs. For example, Google’s search engines are driven by their proprietary Map-Reduce, and Yahoo deploys Apache’s open source Hadoop.

And, 75 years on from Alonzo Church’s pioneering exploration of the entscheidungsproblem[[1]](#footnote-1), λ calculus remains central to Computing education and research. Functional programming is a routine component of many Computer Science undergraduate programmes, as are computability theory, and, to a lesser extent, formal semantics. Functional techniques are being used in novel approaches to constructing scalable heterogeneous multi-processor systems and to developing mission and safety critical systems requiring strong assurances that requirements are met. New calculi building on λ calculus continue to evolve, for example in Milner’s Communicating Concurrent Systems, Π Calculus and bigraphs.

*Content*

Looking at the book from a markedly older and greyer perspective, I feel happy with it, by and large. In particular, I remain firmly wedded to the pedagogy of learning by abstraction from concrete examples, of understanding λ calculus through actually “doing” it in an explicitly operational manner, and of gaining oversight of the layers between a simple, foundational system and a rich language of variegated constructs and structures.

The book’s major eccentricity remains my reformulation of classic λ calculus syntax. In Church’s notation, applications in function bodies are unbracketed, but functions are bracketed except where there is no ambiguity. I chose instead to bracket applications in function bodies but to not bracket functions except where there is ambiguity, as I felt these were more in keeping with programming language conventions. In retrospect, I suspect that this may prove unduly confusing to novices trying to use this book to complement other sources.

I also now think that the account of lazy evaluation could be simplified and tidied up. There is also merit in one reviewer’s suggestion[[2]](#footnote-2) that a pure lazy polymorphic language might have been described given that both Lisp and SML are strict and impure. However, when the book was written, Miranda™ was a commercial product and Haskell had not been standardised.

Finally, my favourite chapter remains that on recursion.

*Conclusion*

When I was wee, my parents had lots of Dover books: Charles Babbage, Lewis Carroll, Gustave Dore... I am tickled pink to join these authors.

So, I would very much like to thank:

* John Crossley for suggesting that I approach Dover;
* John Grafton at Dover for reprinting this book, and for all his support.

Greg Michaelson, Edinburgh, May 2011.

1. Church, Alonzo, *An unsolvable problem of elementary number theory*, [American Journal of Mathematics](http://en.wikipedia.org/wiki/American_Journal_of_Mathematics), 58 (1936), pp. 345–363. [↑](#footnote-ref-1)
2. R. Jones, Times Higher Education Supplement, p29, 29/9/89. [↑](#footnote-ref-2)