

Book Review: “Programming in Haskell” by Graham Hutton

by Duncan Coutts <duncan.coutts@comlab.ox.ac.uk>

Do we need another introductory Haskell book? Is there anything new to be said or a better approach to take? Graham Hutton thinks there is. I think he is right.

Details

Title	Programming in Haskell
Author	Graham Hutton
Pages	171
Published	2007
List price	£23.99, \$45.00, AUD\$79.95 (paperback)
Street price	£22.79, €35,55, 380kr (paperback)
Publisher	Cambridge University Press
ISBN	978-0-521-87172-3 (paperback), 978-0-521-69269-4 (hardback)

About the Author

Dr Graham Hutton is a Reader in Computer Science at The University of Nottingham where he taught the first year functional programming course for several years. He has an impressive publishing record going back to 1990.

Scope

The book covers the Haskell language, programming techniques and the core library functions. It is a tutorial, not a reference manual. It is a pleasantly short book that covers all the essentials and nothing more.

If you don't know Haskell yet, reading this book is probably the quickest and most painless way of getting up to speed.

Audience

The book does not assume any advanced mathematics or much previous programming experience. For those with some background in imperative programming it briefly explains the essential difference in approach. It is perfectly suitable for people who have done a bit of programming and want to know what all this fuss over Haskell is about – you don't need to be a mathematician.

On the other hand it is probably not suitable as a very first introduction to computing. It is not about the very basics of solving problems using algorithms.

Style and approach

The writing style is very clear and straightforward. Examples are used consistently throughout when introducing new ideas. There are no long expanses of exposition. The pace is quite brisk but the explanations are clear and the examples keep the ideas concrete and help build an intuition.

There are exercises at the end of each chapter to help get to grips with the new ideas in the chapter. None of the exercises are essential for understanding the following chapters, so it would be quite possible to skip them on a first reading and come back to them later. There are notes at the end of each chapter giving a bit of context and references to further reading.

It does not deviate into large case studies in the early chapters, instead it uses lots of simple examples at each stage. There is one bigger example at the juncture just before introducing recursion, which is probably appropriate. At each stage we can write and run programs. Each new idea allows us to write more or better code.

The approach of this book is to try to have the reader learn the central ideas of the language as quickly as possible. Remarkably, the core ideas are all covered in the first 70 pages. While it is concise I don't think any essential explanation has been sacrificed, it is just well chosen. The learning curve is not too steep, especially if the reader is prepared to try some of the exercises at the end of each chapter. In the university lecture course upon which this book was based, the students cover roughly the same material in about four months, though it is not the very first course they take.

Comments

The distinguishing feature of this book, in my opinion, is the order and the way in which each topic is introduced. It is clear that Graham has observed how students learn this topic and from that he has carefully considered how to present the material so that it is most effectively learnt. To put it simply, it has been optimised for learning. This is a slightly subtle point perhaps, but an important one. This approach is in contrast to merely presenting the material in a logical order or in a way that reflects the author’s philosophy of functional programming.

For example, I think he is right to introduce types and particularly overloading so early. Similarly, I would say he has made the right decision to defer treatment of lazy evaluation and equational reasoning to the final chapters. IO is presented about halfway through the book, just after the core pure subset of the language.

The presentation can mostly be characterised as a top-down but Graham does explain at various points how newly introduced constructs can be used to explain or give an alternative definition for things that were introduced previously. This should satisfy those readers who understand best when they can see how everything is built from a small set of primitives, but for those who don’t learn that way it does not force that approach upon them. Graham’s approach means that with each new idea we are only taking small steps away from the familiar and the examples allows us to keep an intuition. This contrasts with a bottom-up approach of presenting ideas abstractly and then showing how that allows one to build the familiar.

Calculation

Some presentations of functional programming heavily emphasise the notion that programs are mathematical objects and can be manipulated, optimised and reasoned about using equational reasoning. While this is certainly something we want to tell people, I usually find that the majority of the audience do not naturally warm to the mathematical approach to learning. To many people it seems hard and off-putting.

I believe it is easier for most readers to grasp these ideas after they have got the basics of the language. So it is not that calculational methods are not important, it is just that they are not necessarily the easiest way to learn. For most students, the intuition drives the calculation, not the other way around. It is not obvious in the earlier stages of learning why this “calculation stuff” might ever be useful. There’s the danger that readers and students assume it’s just something academics do so that they have some hard questions to put in exams. I am convinced it is better not to clutter the critical early phases of learning this new language with too much formalism but to explain that later at a more relaxed pace.

Laziness

Similarly, I think it is perfectly OK to defer explaining lazy evaluation. While it is a distinguishing feature of Haskell, you can get quite a long way in learning functional programming before strict or lazy evaluation matters at all. From a practical point of view, the advantage of laziness is mainly that it enable us write programs in a nicer style, which is something that is easier to appreciate once we can read on understand any Haskell programs at all!

IO

IO is introduced with little fuss. Monadic IO may have been remarkable when it was first invented but now it is right to present it reasonably early and as fairly straightforward. Otherwise, the danger is that people are put off by the assumption that IO is hard. On the other hand it is not necessarily great to put monadic effects at the centre of the whole presentation. IO is just something we have to be able to do, it is not an area in which Haskell is particularly radical (except in the way that it cleanly separates it).

A clever aspect of the presentation of IO is that the previous chapter on parsers already introduces the notion of sequencing and the `do` notation (without ever using the term “monad”). So when these concepts get re-used for IO they are already familiar, which makes it clear that this funny `do` syntax is not peculiar to doing IO. It neatly avoids the trap of thinking that Haskell has a special imperative language or that the `do` notation is only for programming with side effects.

Recursion

Graham introduces us to lists, many standard functions on lists and list comprehensions before recursion. This allows the reader to gain confidence with some realistic code before tackling recursion. Then immediately after recursion, we move onto higher order functions. Hopefully this will help readers to not get “stuck” on doing everything with primitive recursion, which is something I see with some of my own students and a phenomenon that Simon Thompson tried to address in the second edition of his book [1].

Chapter summaries

1. The introduction starts with what a function is, what people understand by “functional programming” and how that contrasts with imperative programming styles. It gives a brief overview of the language features that the

remaining chapters cover, a brief historical context of the language and a quick taste of what Haskell code and evaluation look like.

2. After the introduction, the book starts in earnest in chapter 2 by showing some examples of simple expressions on numbers and lists and how to evaluate these using the Hugs interactive Haskell interpreter. It then covers the basic syntactical conventions, in particular the notation for function application which is something that can easily trip up people who are familiar with the convention of other programming languages.
3. The next topic is types: basic types, tuples and lists. It then covers the important topic of function types and currying. Overloading is introduced very early. It then goes on to present the important type classes: `Eq`, `Ord`, `Show`, `Read` and `Num`.
4. Having been armed with the basics of expressions and several useful functions on numbers and lists, the next chapter explains how to build more interesting expressions by defining functions, including conditionals, guards and patterns. After introducing lambda expressions the earlier concepts of function definition and currying are revisited in the light of this primitive.
5. Before getting on to the more tricky concept of recursion, list comprehensions are introduced. Along with the other ideas from the previous chapters we work through a non-trivial example problem: cracking a simple Caesar cipher.
6. The following two chapters cover recursion and higher order functions. Recursion is explained clearly before moving on to examples on lists and examples with multiple arguments, multiple recursive calls and mutual recursion. Since recursion can be a stumbling point for some, the chapter concludes with advice and a five point strategy for making recursive functions. To reinforce the strategy it is demonstrated by working through the process on several examples.
7. In my experience teaching Haskell I have noticed that having discovered and mastered recursion, some students seem to get stuck and write all their code in a primitive recursive style rather than moving on to take advantage of Haskell’s ability to abstract. By presenting higher order functions immediately after recursion, readers of this book should be able to avoid a first order fate. It introduces the idea of functions as arguments and explains `map` and `filter`, as always, with plenty of examples. We are introduced next to `foldr` and `foldl` and shown how they can be used to capture the pattern of recursion used in many of the simple recursive functions defined previously.

8. The first major example after covering the core language is a parser for a simple expression language. It shows off the ability to create abstractions and, importantly, it introduces the (`>>=`) sequencing operator and the `do` notation.
9. Sequencing and the `do` notation is immediately re-used in the next chapter in explaining interactive programs and IO. It is illustrated with a console calculator and game of life program.
10. The first of the more advanced topics is on how to declare new types, first type aliases and then new structured data types. It covers recursive data types including trees and a bigger example using boolean expressions. Significantly, it covers how to make new types instances of standard type classes such as `Eq`. It gives examples of how to define instances using both `instance` declarations and the `deriving` mechanism. It also comes back to sequencing, showing how the previous examples of parsers and IO fit into the `Monad` class, which makes it clear when the `do` notation can be used.
11. The next chapter is dedicated to a case study working through Graham's famous "countdown" program. It covers how to model the problem, starts with a simple brute-force search algorithm and then refines it in several stages to make it much faster. The solution makes good use of the features introduced earlier like list comprehensions, standard list functions, recursion and tree data types.
12. The penultimate chapter covers lazy evaluation. It compares 'inner', 'outer' and lazy evaluation strategies on the grounds of termination and reduction count. It continues with examples of how it allows infinite structures and helps modular programming. It also covers strict application (mentioning `foldl'`) and when it is useful to control space use.
13. The final chapter is on reasoning about programs. It starts with equational reasoning and continues with inductive proofs on numbers and lists. In addition to proving properties about programs it shows how the equational approach can be used to optimise programs by deriving an efficient implementation from a simple initial implementation. It uses the nice example of eliminating expensive `append` operations by abstracting over the tail of the list, obtaining a rule and applying the rule to get a new efficient version. It's a convincing advertisement for the formal viewpoint: it gives a huge speedup and a final implementation that we would not necessarily have thought of directly.

Criticism

Apart from annoyances noted below, personally, I find little to complain about.

It is possible that instructors who consider using the book in a lecture course might worry that the examples are not sufficiently exciting or that there may be insufficient motivation for why we might want to do things like parsing. I think the examples are about right in size for illustrating the ideas, bigger more exciting programming tasks can always be set for practical labs.

One may complain that the book does not cover enough, that we might prefer more topics on more advanced material. The topics selected cover an essential minimum, which seems like an appropriate place to cut. Probably the topics that will be most missed are operational issues like space and time complexity. There is certainly a gap in the market for an intermediate Haskell book that starts where this one leaves off.

One might also complain that not every aspect of the language is covered. It uses the common declaration style with named functions and `where` clauses, it does not cover `let` expressions and `case` expressions are only briefly mentioned. There are various other lesser used corners of the language not mentioned. The book is very clearly presented as not being a language reference manual so it is hard to make this kind of criticism stick.

Annoyances

One minor thing to watch out for is that while the code examples are beautifully typeset using `lhs2TeX`[2] they cannot be directly typed into Hugs because many of the operators are typeset as symbols that do not correspond to the ASCII syntax that one has to use. The section introducing the Hugs system and the prelude references an ASCII translation table in appendix B. I would have preferred if this table were just included inline, like the table of hugs commands. I managed to miss the reference on a first reading.

Unfortunately, a few of the code examples rely on bugs in an old version of the Hugs system that have since been corrected. For example, Hugs previously exported several non-standard and `Char` module functions through the `Prelude`. After Hugs was fixed the three other books published around that time suffered from this problem. It is rather a shame that this new book suffers the same problem. While the solution in each case is very simple, e.g. just import the `Char` module, these kind of minor problems can be quite bewildering to new users. An errata with all the details is available online [3].

A slight quibble is that the introduction describes the Haskell98 standard as being “recently published”. It is true that the final version of the language report was only published in 2003, however an 18 year old university student is likely to

claim that anything that is apparently nearly 10 years old is archaic. For marketing purposes it is probably better to describe Haskell98 as the stable version of the language. There is an obvious tension in wanting to make it clear to potential readers that the material has not been superseded by changes in the language, but at the same time not give the impression that they would be learning something that is dead.

Comparison with other books

There are three main other Haskell books available in English. There are also recent books in Portuguese, Spanish, Romanian, Russian and Japanese [4].

It is an interesting sign of the times perhaps that the book is titled simply *Programming in Haskell* rather than the title having to relate Haskell to functional programming in general.

Introduction to Functional Programming using Haskell

The Introduction to Functional Programming using Haskell [5] is the text for the mathematically minded. It aims to teach the concepts of functional programming. To be concrete it uses Haskell. The emphasis is on constructing functions by calculation. It introduces equational reasoning very early where as *Programming in Haskell* delays that topic to the final chapter. Conversely, IO is deferred to the final chapter where as *Programming in Haskell* covers it much earlier.

It is a much more substantial book (448 pages) and covers topics like trees, abstract data types and efficiency/complexity.

The Haskell School of Expression: Learning Functional Programming through Multimedia

In a complete departure from the mathematical approach, *The Haskell School of Expression* [6] takes the somewhat radical approach of using multimedia as the main motivating examples throughout the book. This is an approach that appeals to some, but not to others.

It starts at a somewhat more advanced level than *Programming in Haskell* and goes on to cover modules, higher order and polymorphic functions, abstract data types, IO and various domain-specific embedded languages.

Again, it is considerably longer at 382 pages.

Haskell: The Craft of Functional Programming

Simon Thompson’s *Haskell: The Craft of Functional Programming* [1] sits somewhere between the other two in terms of style. It aims to teach Haskell and the functional programming approach. As of the second edition it uses more examples and more of a top down style, where for example, functions are used before going into the details of their implementation.

It covers modules reasonably early where as *Programming in Haskell* does not cover them at all. It introduces proofs and equational reasoning much earlier. It is much less compact in its explanations than *Programming in Haskell* and is a much larger book (512 pages). It does cover several more advanced topics such as algebraic data types and time and space behaviour. It defers IO to a much later chapter than *Programming in Haskell*.

Conclusion

In my opinion, this book is the best introduction to Haskell available. There are many paths towards becoming comfortable and competent with the language but I think studying this book is the quickest path.

I urge readers of this magazine to recommend *Programming in Haskell* to anyone who has been thinking about learning the language.

The book is available now from online [7] and high street bookstores.

Acknowledgements

Thanks to The Monad.Reader editor Wouter Swierstra for arranging a review copy of the book for me. Thanks to Lennart Kolmodin and other denizens of #haskell for helpful comments on drafts of this article.

About the Reviewer

Duncan Coutts has a BA in Computation from the University of Oxford. He is still there trying to finish his PhD. He has seen Haskell teaching from both sides: as an undergraduate he attended Richard Bird’s first year functional programming course; as a teaching assistant he now runs the practicals for the same course.

References

- [1] Simon Thompson. **Haskell: The Craft of Functional Programming**. Addison Wesley, 2nd edition (1999).
- [2] Andres Löh. lhs2TeX. <http://www.informatik.uni-bonn.de/~loeh/lhs2tex/>.
- [3] Errata. <http://www.cs.nott.ac.uk/~gmh/errata.html>.
- [4] http://haskell.org/haskellwiki/Books_and_tutorials#Textbooks.
- [5] Richard Bird. **Introduction to Functional Programming using Haskell**. Prentice Hall Press, 2nd edition (1998).
- [6] Paul Hudak. **The Haskell School of Expression: Learning Functional Programming through Multimedia**. Cambridge University Press (2000).
- [7] Programming in Haskell website. <http://www.cs.nott.ac.uk/~gmh/book.html>.