either the subcomponent interfaces or the rest of the program. If the program is declarative, this is impossible, since the only way is to thread an accumulator through the program.

The proposed solution runs entirely against the orthodoxy that compositionality is necessary for modularity, but it is persausive. Much to their credit, the authors have robustly defended this view and others in several online forums, and have shown they are prepared to accept and develop viewpoints distinct from their own.

In closing, I must ask the inevitable rhetorical questions: would CTM have been better as a series of smaller works? – and when will we see a book-length treatment of secure distributed systems programming in Mozart/Oz?

Thanks to Tim Bourke, Gregoire Hamon, Ben Lippmeier and Bernie Pope for helpful feedback on this review.

References

Abelson, Harold, & Sussman, Gerald J. (1996). *Structure and interpretation of computer programs*, 2nd edition. The MIT Press.

Alice ML. (2007). *http://www.ps.uni-sb.de/alice/*.

Bird, Richard, & Wadler, Philip. (1988). *Introduction to functional programming*. Prentice Hall.

Dijkstra, Edsger W. (1976). *A discipline of programming*. Prentice Hall.

Peyton Jones, Simon. (2003). *Wearing the hair shirt: a retrospective on Haskell*. Invited talk at 30th ACM Symposium on Principles of Programming Languages.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press.

Roy, Peter Van. (2003). *http://lambda-the-ultimate.org/classic/message9361.html*.

Saraswat, Vijay A. (1993). *Concurrent constraint programming*. MIT Press.

Peter Gammie
School of Computer Science and Engineering, The University of New South Wales

Programming in Haskell by Graham Hutton, Cambridge University Press, 2007, 184 pp., ISBN 0-521-69269-5.
doi: 10.1017/S0956796809007151

Though functional programming is still far from mainstream, the growing popularity of languages such as Haskell has inspired many authors to guide a variety of readers into the paradigm. *Programming in Haskell* is one such book, serving as an introduction to Haskell for audiences with little to no prior knowledge of programming.

In 2007, Dr Graham Hutton wrote *Programming in Haskell* for the Cambridge University Press. A reader in computer science at the University of Nottingham, where he helps to lead the Functional Programming Lab, Dr Hutton has over 15 years of experience in researching functional programming languages and over 10 years of experience in teaching Haskell in particular. His experience is evident in the excellent structure of the book, ordering chapters and concepts carefully to make the transition into functional programming as smooth as possible.

The paperback book is a slim 184 pages, with wide margins to write solutions to simpler exercises and clarifications. It is reminiscent in style and form to familiar books like *The C Programming Language* (Kernishan & Ritchie 1988) and *The UNIX Programming Environment* (Kernighan & Pike 1984), maintaining a personal touch with a lean style. The text is more concise than almost all other available tutorials, which has been made possible through the examples the author provides.

The book is divided into 13 chapters and 2 appendices. It is written as a tutorial and, as stated before, assumes no prior knowledge of programming, though the book is much easier to digest for students who have at least been exposed to the vocabulary of programming. Because of its tutorial style, each chapter surveys a new aspect of Haskell, not enough to serve as a reference but in sufficient detail to subsequently allow the student to tackle denser documents about Haskell, such as language specifications, which are often referenced at the end of each chapter. In this way, the book serves as an excellent map to Haskell resources online, for hungrier students.

Each chapter provides around five exercises, with just the right amount of difficulty to solidify the surveyed concepts without disconcerting the reader. Although they serve as a good starting point, most of the exercises in the book are not challenging enough on their own – professors and autodidacts should find more challenging exercises from other resources in order to master the material being taught. In this way, the exercises are reminiscent of the UNIX and C manuals mentioned earlier. Fortunately, in the chapter on input/output (IO), the author provides more difficult exercises and projects to allow the students to become comfortable with the topic.

Where this book excels is in the order and style of its exposition. The first seven chapters are excellent in their concise delivery. The author orders the concepts to make learning as easy as possible for people with little exposure to any sort of programming. Within a few hours of self-study and toying with Hugs, students can become comfortable with several fundamental concepts of Haskell, such as functions and basic types. The author suggests in the introduction that the book can be covered in 20 lecture hours, supplemented with 40 hours of private study. This is an upper bound – students with prior programming experience will likely cut through the book much faster.

The book opens with an introduction to functions. The author immediately contrasts functional programming to imperative programming, comparing the verbosity of the two different approaches in summing the first n integers, and continues with the standard example of quicksort, describing briefly the algorithm as well as its Haskell implementation.

The second chapter introduces the reader to coding in Haskell, describing how to use the Hugs system with a brief and well-placed digression on function application. He then describes the miscellany of coding, such as scripts, naming requirements, code layout and comments.

The third chapter introduces types and type classes, first with basic types, moving to lists, tuples, functions, currying, polymorphic and overloaded types, closing with basic type classes. The fourth chapter continues with more fundamental notions, such as conditional expressions, guarded equations, pattern matching, lambda expressions and the sectioning of operators.

The next two chapters introduce list comprehensions and their attendant syntax, such as generators and guards, and provide the first of several extended examples which serve to motivate the reader and unify the covered concepts – in this chapter, a statistical way to crack the Caesar cipher. Here, the author introduces list comprehensions before recursion, a pedagogical decision shared by Wadler (Richards 1998).

The sixth and seventh chapters delve deeper into functions, introducing recursion with a brief tutorial on how to write recursive algorithms, a common stumbling block for introductory computer science students. The author continues to higher order functions, spending several pages on left and right folds, another common point of confusion. After describing function composition, the author provides another motivating extended example of a binary string transmitter, with error-correction as an exercise.

Often, the major question in the study of Haskell pedagogy is how to teach monads (Jones 2007). Haskell's purity is one of the language's greatest strengths, allowing for abstract and powerful generics, but this power is largely invisible to people starting out in the language, and the resulting theoretical constructs – in particular, the notion of monads for state management – serve as the steepest part of the learning curve.

Accordingly, this is the most confusing part of the book. The author reserves the word *monad* for the eighth and ninth chapters, addressing abstract monadic structures from the point of view of parsing. He briefly describes what parsing is and then spends the chapter sketching Haskell for the task. In this way, he introduces monadic notions such as sequencing without explicitly referring to them as monadic. He continues with an extended example of parsing arithmetic expressions, which transitions into introducing input and output. The author provides a brief analogy of monads as 'states of the world' and continues his extended example by providing a calculator interface to the parser. He then provides another extended example of the Game of Life, with a description and code for an interface.

The author is not alone in his decision to push IO and monads to the end of his book – other authors of introductory Haskell texts, such as Thompson (1999), do the same. The most notable example of the opposite structural choice, introducing IO and monads early, is in Hudak's (2000) *Haskell School of Expression*. However, Hudak's book is example-driven, whereas *Programming in Haskell* uses examples to illustrate described concepts, so the choice to relegate monads to the end of the book makes sense (Sydow 2000). Although it may have cut into the flow of the first seven chapters, it would have been better if the author simply introduced monads upfront and then tackled the parsing task. For this reason, and because the code is not self-contained, the two chapters on parsers and monads are the most confusing of the book. Readers might want to turn to other sources to learn about monadic structures. However, this should not deter potential readers. Excluding the section on parsers and monads, the author exposes all other concepts with clarity and concision.

After this, the author returns to his effective style. The rest of the book consists of several extended examples, starting with a simple tautology checker to illustrate defining new data types, as well as an abstract machine to control order of evaluation. After explicitly mentioning monads to unify what has been covered, the author introduces type instance declarations. Another example, to illustrate the process of algorithmic problem-solving, is the countdown problem (as in the British television programme), tuning a naively inefficient solution in several pedagogic steps. He then covers lazy and strict evaluation, termination, modular programming and infinite structures. Finally, the last chapter describes equational reasoning, introducing induction and applying it to two extended examples – eliminating the use of append in certain cases and verifying compiler correctness.

Two groups of people must consider this book. The first is professors interested in rapidly introducing students to fundamental concepts in functional programming. This book, supplemented with online resources and professorial guidance, could easily serve as the textbook for a semester-long course on functional programming. The second group is programmers interested in surveying the functional paradigm as quickly as possible. The various examples in the book are excellent to see at a glance how the functional approach differs from the imperative one, especially with the countdown problem, and are likely to inspire readers from an imperative background to learn more about functional programming. Other groups of people may not find anything new in its contents but will benefit from its excellent didactic style.

With its ripe selection of examples and its careful clarity of exposition, the book is a welcome addition to the introductory functional programming literature.

Saketh Bhamidipati
Cambridge, Massachusetts, USA

References

Hudak, P. (2000) *The Haskell School of Expression: Learning Functional Programming through Multimedia*. New York: Cambridge University Press.

Jones, I. (2007) Review of Hudak (2000). *J. Funct. Prog*, **17**(3), 426–428.

Richards, Jr., H. (1998) Review of Thompson (1999). *J. Funct. Prog*, **8**(6), 633–637.

Kernighan, B. W. & Pike, R. (1984) *The Unix Programming Environment (Prentice-Hall Software Series)*. Prentice Hall, New York.

Kernighan, B. W. & Ritchie, D. M. (1988) *The C Programming Language*, 2nd ed. Prentice Hall PTR, New York.

Thompson, S. (1999) *Haskell: The Craft of Functional Programming*, 2nd ed. Addison Wesley, London.

Von Sydow, B. (2000) Review of Hudak (2000). *J. Funct. Prog*, **10**(5), 501–508.

Programming Erlang – Software for a Concurrent World by Joe Armstrong, Pragmatic Bookshelf, 2007, p. 536. ISBN-10: 193435600X.
doi:10.1017/S0956796809007163

Context

A funny thing happened on the microprocessors' race to faster and faster clock speeds; somewhere along the way, the speeds and feeds hit a brick wall and the only way out is lateral – more cores rather than faster CPUs. It started with hyper-threading and progressed to dual cores and to multi-core CPUs. While this paradigm shift solved the hardware challenges, a new bottleneck arose: in the software! Multi-threaded programming, which was mainly for leveraging I/O waits and SMPs could not scale to moderately massive parallelism; while the domain complexity of concurrent programming itself is interesting, the additional accidental complexity of threads and murexes and semaphores in traditional programming just does not cut it.

The answer it seems is 'Pluralitas non est ponenda sine neccesitate', i.e. the singularity of immutability and statelessness of functional languages rather than the pluralities (of mutexes, semaphes, spin locks and code locking): a manifestation of Ockham's Razor! As a result, more and more folks are looking towards functional programming as the silver bullet for providing the software version of Moore's Law! In fact a timely article in Dr. Dobb's Journal (Swaine 2008) wonders whether 'functional programming is on the verge of becoming a must-have skill'! Erlang is at the forefront of this revolution – an article in ACM Queue (Larson 2008) is of the opinion that '… designed for concurrency from the ground up, the Erlang language can be a valuable tool to help solve concurrent problems'.

In short Joe Armstrong's book *Programming Erlang – Software for a concurrent world* is at the right place and at the right time! As the title implies, the focus of the book is more on concurrency than functional programming but functional programming is the essential backdrop for the functionality required.

Overview

This is not a traditional programming book – rather I consider this to be a systems book, and that comes from the fact that it is difficult to separate 'Erlang the language' from 'Erlang the system'. The book follows the architecture of the Erlang system, which makes perfect sense, especially as the author is one of the originators of the language.

The book is pretty thick – 20 chapters and 5 appendices, coming out at over 500 pages. The subject is deep but the style makes it interesting and easy to comprehend. I felt that the book ended very fast! The book has a logical progression and is logically divided into four parts:

- Part 1 covers the important concepts of the language with enough syntax to get started, and ends with an overview of compiling and running Erlang Programs.