In summary, I highly recommend *Software Abstractions: Logic, Language, and Analysis* to anyone with an interest in modeling and analyzing software. It is suitable for both class-room use and for reference long after the basics have been mastered. Systems like Alloy should be in the toolbox of all software designers and developers, so such a comprehensive book on this topic is very welcome.

Anthony M. Sloane
Macquarie University, Sydney, Australia

Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004. ISBN: 0262220695 Price $70. 930pp.
doi:10.1017/S0956796808007028

I came to CTM, as this book is familiarly known, with a deep appreciation for the innovative contribution Peter Van Roy made towards implementing logic programming systems in the early 1990s. It is good to see him and his collaborators continue to push the frontiers of this tradition, and making that work accessible to the masses through this book.

CTM is of similar stock to such rigorous introductory textbooks as the classics by Abelson and Sussman (1996) and Bird and Wadler (1988), and is significantly less formal than any of Dijkstra's classics (1976). In contrast to these texts, the main theme of the book is concurrency from a systems engineering perspective, culminating in discussions of three application domains: graphical user interfaces, distributed programming and constraint programming. The reader is expected to have a reasonable grasp of the basic techniques of sequential programming, and so this book complements most other in-depth programming texts.

The target audience, late-undergraduate or early-postgraduate students, may find some of the introductory material a bit patronising. It is unfortunate that while the book is substantially about concurrency, it is itself mostly sequential-access: the reader will find it necessary to carefully peruse these early sections in order to grasp the syntax and semantics of Mozart/Oz, the programming language at the centre of the CTM world view. Some of this tedium is alleviated by the delightful ease of experimenting with the mature Mozart implementation.

Formal operational semantics are provided for the various 'kernel languages' that are used to explain language features, ultimately collected and distilled in the relatively technical Chapter 13. Readers of TaPL (Pierce, 2002) will be familiar with this approach, although here the semantics is given in the style of a concurrent constraint language (Saraswat, 1993). By itself it would be difficult to credit these sections as a sufficiently broad introduction to programming language semantics, for no properties are established. Also it is unfortunate that the Hoare logic so clearly presented in Chapter 6 is not formally related to the ongoing operational story.

At the core of the Oz approach is the *dataflow variable* (also known as the *declarative variable*), an object that can be declared in one scope and bound in another. Prolog programmers will be on familiar ground with their use in difference structures (Section 3.4.4), and in the underpinnings of the declarative concurrency development (Section 4.3). While this style of concurrency requires linguistic support to be completely natural, there are library-based implementations in various languages that embody the abstraction.

CTM has the clearest presentation of declarative programming (broadly taken) that I have yet found; the benefits for program structure and reasoning are strongly articulated and beautifully illustrated, and the limitations are carefully teased out. The presentation of declarative concurrency is a highlight of the book, and as the authors observe, deserves to be much more widely understood and applied. To a functional programmer it is somewhat

reminiscent of coming to terms with the lazy functional programming tradition's attachment to purity (Peyton Jones, 2003).

The book continues with more traditional forms of concurrency, where threads communicate by passing messages or sharing state, with dire warnings about the difficulties in doing the latter. There are extended discussions of object-oriented and relational programming before the three case studies are presented.

As Oz is a dynamically typed language, types get short shrift. I found this to be somewhat of a hole in the promise to cover all major concepts and paradigms, despite the occasional nod to the typeful community. Perhaps the biggest problem with not having an explicit language for types is the difficulty this causes in discussing them; some interfaces would be a lot easier to fathom if a signature with types could be used, instead of a page or so of text. This perspective is tacitly acknowledged in Section 3.4.1, which informally presents a notation similar to the ubiquitous Algebraic Data Types for the purposes of explaining recursion over structured data.

I observe that those who do want a statically typed language with similar features to Oz can investigate Alice ML (2007), which also runs on the Mozart system. There *promises* stand in for dataflow variables, and the base language is Standard ML.

A central tenet of CTM is that a *multi-paradigm* language is of great utility, almost of necessity, in writing large pieces of software. While this approach allows the authors to express a wide variety of concepts in a series of closely related kernel languages, it also implies that traditions lose their moorings to some extent. For example, those who are accustomed to typeful thinking or top-down design may wish that discussions of examples started with ontological rather than algorithmic concerns (such as the part on transactions, Section 8.5), and to be able to determine that a piece of code is declarative without having to examine it in its entirety. It also leads to a certain amount of semantic hair splitting over how faithfully the paradigms are presented.

Similarly there are limits to how far any given paradigm can be naturally expressed in Oz; for example, CTM gives a good overview of higher-order programming (Section 3.6) but does not use the technique as pervasively as a functional programmer would; there is no development of parsing combinators, monads, continuations and so forth. This partially reflects the authors' biases, and that Oz, to a similar but lesser extent than C++, allows but does not encourage the use of high-order idioms; currying is manual, function abstraction is a tad verbose, the type system provides little assistance, and as observed above, effects are unencapsulated.

This approach also leads to long threads of discussion that span several chapters, such as the presentation of the various notions of interfaces and implementations. Ultimately the authors are to be applauded for placing these topics in a common language, but the structure of the book limits its utility as a reference work. This is partially remedied by the historical perspectives given in some chapters.

One might quibble over how strongly the multi-paradigm approach is validated by CTM, given that several application domains – artificial intelligence, compilers and language processors, numerical algorithms, sophisticated data structures, to name a few – are not treated. I readily grant that there is a water-tight omitted-for-reasons-of-space argument that can be marshalled here.

The overarching achievement of this book is to be so provocative that one wants to engage the authors in debate about almost everything they say. Partly this is due to the chirpy writing style that gives one the feeling that the authors would much prefer to be pair-programming with oneself and each of the other readers than using this unidirectional medium. Also this is partly because some of their arguments are difficult to evaluate, but mostly it is their delicious iconoclasm. Take, for example, their assertion that state is necessary for modularity (Section 4.8.2):

[Instrumenting a program.] We would like to know how many times some of its subcomponents are invoked. We would like to add counters to these subcomponents, preferably without changing

https://doi.org/10.1017/S0956796808007028 Published online by Cambridge University Press

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.