J. Functional Programming **9** (4): 479–482, July 1999. Printed in the United Kingdom

Book reviews

The Functional Approach to Programming by Guy Cousineau and Michel Mauny, Cambridge University Press, 1998, ISBN 0-521-57681-4 pbk, xiv+445pp.

The *Functional Approach to Programming* has been written for a fairly sophisticated audience with a good understanding of programming (but not necessarily functional programming), data structures, compilers, semantics and computer graphics. The book originated from material for a course by one of the authors. It covers a wide range of topics, but in spite of the size of the book I found the coverage sometimes a little unsatisfactory because (1) a fair number of the programming examples are not discussed in depth, (2) some of the design decisions are not well motivated, and (3) there is no emphasis on program design.

This is a professional English translation of a French book. Unfortunately, some of the jargon has been literally translated. For example, the book uses type synthesis where type inference would be more appropriate, convergent instead of confluent, delayed evaluation instead of normal order evaluation, ascendant analysis instead of bottom-up analysis, etc. There are indications that the programming examples have also been translated with the risk of introducing errors. A related issue is the choice of names for certain standard operations. For example, where an English text would use `reduce` or `foldr`, the *Functional Approach to Programming* uses `list_it`.

The book has 13 chapters, which are grouped into three parts as discussed below. The text provides some exercises, with a pointer to the authors' web site for answers. The bibliography is adequate, but could have been more extensive. Each chapter is concluded by a summary and a section entitled 'To learn more'. I found this a nice feature of the book.

Basic principles

Chapter 1 covers in less than 30 pages everything from the syntax of CAML to higher order functions, polymorphism, type inference and the equivalence of functions. CAML is rather similar to SML except that CAML lacks the sophisticated module facility provided by SML. As a consequence most examples are fragments of programs, rather than collections of modules that form a program. It is noteworthy that only in Chapter 12 are there pictures showing, for example, how the append function on lists introduces sharing. In my experience, these pictures are an invaluable pedagogical aid in explaining to most students from the beginning what their programs are really doing.

Chapter 2 introduces elementary data structures (tuples, lists, trees, sets), as well as some useful operations of the data structures, like Quicksort. The fine points of parsing strings into CAML data structures and printing the data structures again are not explained.

Chapter 3 presents the denotational and operational semantics of the core of CAML, assuming that the reader is familiar with formal semantics. I enjoyed reading this chapter, as it takes a rather more fundamental approach than most other texts that I know of. The fundamental approach taken could have been pushed a little further by discussing the soundness of the type system more fully.

Chapter 4 introduces the imperative features of CAML. These include mutable records, mutable references, side effecting I/O and 'physical' equality (used to detect sharing). The

chapter tries to extend the semantics of the core language already given in Chapter 3 with a semantics for the imperative features, but falls short because the effects of imperative constructs on the state of the computation are not modelled.

Applications

Chapter 5 introduces the first application: unification over terms. This is an interesting problem, but as an application it is rather closely related to what implementors of functional programming languages are interested in. I felt that the use of exceptions in this chapter (see, for example, `assoc` on page 143+) is a little unnatural.

Chapter 6 discusses trees represented as data structures. Basic algorithms are presented for building and searching binary trees and AVL trees. The algorithms over trees are then used to implement dictionaries, sets and queues. The worst case complexity of the algorithms is discussed, but the more important amortised complexity is ignored. Okasaki's book *Purely Functional Data Structures* treats this subject in a more systematic way.

Chapter 7 discusses graphs, but instead of implementing the graphs as data structures, they are implemented as functions. I found this a creative approach, which is shown to have useful applications to programming simple games, like solitaire. It would have been interesting to see how elegant CAML, with its range of imperative features, could have been used to implement graph traversal in the more traditional way. I found the compact encoding of the state of the games somewhat *ad hoc*. For example, on page 219 we find a fairly inelegant way of compressing 33 bits into two integers.

Chapter 8 discusses another favourite of implementors of functional languages: parser combinators. These make it possible to write a parser in such a way that it can be read as a grammar. Comparing the power of the parser combinators presented here with Wadler's list-of-successes method left me a little less than impressed. For example, when written using Wadler's parser combinators, the grammars $(a \mid \epsilon)a$ and $aa \mid a$ behave indistinguishably. Not so in the approach taken here.

Chapter 9 presents the MLgraph library and its applications to geometric drawings. The first application is rendering binary trees in an aesthetic fashion. The more demanding problem of rendering graphs is not discussed. The second application is tiling in the hyperbolic plane. The images generated are truly amazing, resembling M. C. Escher's engravings. This is clearly a very strong point of the book.

Chapter 10 discusses exact arithmetic over natural, integer and rational numbers. In the first instance, arbitrary numbers are implemented as lists of base 10,000 numbers. Secondly, an implementation is given using doubly-linked lists built out of mutable records. This is followed by an application showing how π can be approximated using a series expansion. It would have been useful to give some ideas on the complexity of the various operations.

Implementation

Chapter 11 presents an environment-based, eager evaluator of the purely functional core of CAML. The implementation is directly based on the operational semantics presented in Chapter 3. The implementation is then changed to provide lazy evaluation, unfortunately without also showing the changes in the operational semantics.

Chapter 12 introduces the idea of compiling the purely functional core of CAML into an abstract machine code. The semantics of the abstract machine instructions are presented using an *ad hoc* notation rather than the more usual operational semantics notation. The compilation of core CAML to abstract machines is done in the usual way, using compilation schemes.

Chapter 13 revisits the type inference system from Chapter 5. Type schemes are introduced

https://doi.org/10.1017/S095679689921355X Published online by Cambridge University Press

to avoid having to type check a let expression once for each of its uses. Good use is made of mutable record to improve the efficiency of the unification process.

Summarising, this is an interesting book on 'functional programming', which covers a lot of ground. Contrary to what the title suggests, the book is not so much about 'programming'. I feel that the authors could have done more to show case the strengths of functional programming, while making use of CAML's imperative features where necessary to achieve efficiency. The book is a translation, which has inevitably reduced the quality somewhat.

Pieter Hartel
University of Southampton, UK

Computability and Complexity: From a Programming Perspective by Neil D. Jones, MIT Press, 1997.

Neil Jones is well known as a programming languages researcher, but he is equally at home in the theory community. His new textbook, *Computability and Complexity: From a Programming Perspective*, is one which perhaps only he could have written.

As the title suggests, Jones approaches the fields of computability and complexity from a programming language perspective. That is, he takes simple programming languages as his basic models of computation, rather than the ubiquitous Turing machine. Not only does this allow the reader to more easily draw on all her hard-won intuitions about programming in understanding computability and complexity, it also allows the reader to more easily apply results from computability and complexity to everyday programming tasks.

Most of the book is formulated in terms of a toy imperative language with sequencing, assignments, and `while`-loops. This is already a big improvement over the finite state machines used to express control in Turing machines. Of course, control is only half the story – one must also consider the kinds of data manipulated by the programs. Here, Jones makes another huge improvement over typical presentations. Rather than choosing an unstructured domain such as symbols or integers, he allows structured data in the form of Lisp s-expressions. This inspired choice really pays off in the study of computability, where many of the important results involve encoding programs as data. As you can probably imagine, it is much easier and much more intuitive to encode a program as an s-expression than to use the more common Gödel numbering.

Jones balances his unconventional presentation with a fairly conventional choice of topics. The first half of the book covers computability theory, including standard topics such as decidability and the halting problem; the robustness of computability over various models of computation; Rice's theorem; and Gödel's incompleteness theorem. The second half of the book covers complexity theory, including the usual range of complexity classes such as PTIME, NPTIME, PSPACE, and LOGSPACE; completeness results; Levin's theorem; and Blum's speedup theorem. My only complaint in these areas is the skimpy coverage of NP-completeness. Most readers will probably want to supplement the text with additional references such as Garey and Johnson's classic *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

Jones also includes extensive discussion of several non-traditional topics related to his own personal research. For example, programming language researchers will not be surprised to find a chapter on partial evaluation. Similarly, complexity researchers will welcome Jones's proofs that constant factors do matter for language-based models of computation (whereas constant factors do not matter for Turing machines), and that the complexity classes LOGSPACE and PTIME can be precisely characterized as the sets of problems solvable by programs written in certain restricted languages.

The greatest weakness of this book is that it was released a few rounds of proof-reading too early – it is littered with more mistakes than I have ever before encountered in a published

https://doi.org/10.1017/S095679689921355X Published online by Cambridge University Press

book.[1] Most of the mistakes are minor and easily corrected, but they are so numerous that they are practically guaranteed to confuse the unwary or immature reader.

In spite of the mistakes, I highly recommend this book to programming language researchers as a reference or as a text for self-study. Unfortunately, I cannot recommend it as a textbook for undergraduate students or beginning graduate students, at least not yet. I am hopeful that most of the mistakes will be corrected in a second edition – perhaps even in a second printing – and I would be pleased to recommend the book for classroom use at that time.

Chris Okasaki
Columbia University

[1] An extensive list of the most important technical corrections is available from the author's web site at `http://www.diku.dk/users/neil/Comp2book.html`.

https://doi.org/10.1017/S095679689921355X Published online by Cambridge University Press