J. Functional Programming **8** (6): 633–637, November 1998. Printed in the United Kingdom

Book reviews

Haskell: The Craft of Functional Programming by Simon Thompson, Addison-Wesley, 1996.

Miranda 81: The Craft of Functional Programming by Simon Thompson, Addison-Wesley, 1995.

These two textbooks are intended for introductory programming courses using Miranda* or Haskell. Although they have no substantive prerequisites, and are therefore accessible to novices, they are clearly designed for the serious computing-science student. Used in a first programming course, they lay the groundwork for a comprehensive undergraduate program. Moreover, they contain plenty of material to interest advanced undergraduates or even beginning graduate students.

The two books are very similar, and could reasonably be viewed as Miranda and Haskell editions of the same book. In this review, a reference to the book in the singular refers to both books, unless one of them is specified explicitly.

The text is divided into three parts. Part I, 'Basic Functional Programming', aims to "build a foundation, focusing on programming over basic types and lists, using first-order, non-polymorphic programs". This part's five chapters present several moderate-sized case studies, including a simple library database, a supermarket-bill exercise, and a text-layout program. These are big enough to be interesting, but not so big as to overwhelm the novice. It's hard to imagine tackling such problems in the first few weeks of a course using Pascal or C.

Two of the chapters introduce proofs by induction over natural numbers and lists. Formal verification is (finally) beginning to take hold, in both industry and academia, and functional programming is an ideal arena in which to introduce it. My students have mostly taken to formal proofs enthusiastically; one senses a feeling of relief that a proof's correctness can be determined objectively, and does not depend on the whims of a grader.

Another valuable feature of Part I is its continual emphasis on *calculation*, which is the author's term for textual expression evaluation, i.e. string reduction. Practice in calculation dispels a substantial element of mystery from students' understanding of how computation 'works', and gives them a valuable tool for comprehending and designing functions and algorithms. As an expository device, the only problem with pure string reduction is that it obscures the full laziness of graph reduction. Much later in the book, Thompson models graph reduction's property that 'a duplicated argument is never evaluated more than once' in calculations by 'doing the corresponding steps simultaneously', as in

$$\begin{array}{rcl} \text{let } f\,x & = & x + x \text{ in } f(9-3) \\ & = & (9-3)+(9-3) \quad \text{-- two subtractions count as one} \\ & = & 6+6 \\ & = & 12 \end{array}$$

This mechanism depends on the student's understanding of just when such simultaneous steps are legitimate, and therefore often leads to confusion.

* Miranda is a trademark of Research Software Ltd.

The instructor can clarify the nature of full laziness (without taking on the typographical and pedagogical complications of directed graphs) by binding arguments to parameters by their names:

$$\begin{aligned} \text{let } f\,x &= x + x \text{ in } f(9-3) \\ &= x + x \text{ where } x = 9 - 3 \quad \text{-- only one subtraction} \\ &= x + x \text{ where } x = 6 \\ &= 6 + 6 \\ &= 12 \end{aligned}$$

Here the role of graph reduction's pointers is played by the parameter names. The result is textual calculations of the same complexity as actual graph reduction (a formal treatment of this problem is given in Ariola *et al.* (1995)).

Part I contains 164 homework problems, ranging from small limbering-up exercises to problem sets that combine to form substantial projects. Used carefully, they can last for several semesters before repeating.

Each of the two long chapters devoted to program construction ends with a nice set of design guidelines distilled from the preceding material. The first of these sections, which precedes the introduction of lists, necessarily focuses on small-scale issues of function design, but nevertheless manages to stress the importance of the *divide and conquer* strategy. The second design-guidelines section emphasizes the inevitability of change, and the necessity of planning for it. It also introduces a thread on error handling which is revisited several times, with increasing sophistication, later in the book.

Part II is entitled 'Abstraction'. Its four chapters introduce the aspects of Haskell that show the student with some conventional programming experience the first substantial payoff for her efforts in adapting to the functional paradigm. To quote from the introduction, "Only when readers are fluent with the basics of functions, types, and program construction do we look at the three ideas of higher-order functions, polymorphism, and type classes...".

The material proceeds in an orderly fashion, building on the foundation laid in Part 1. List-processing functions are generalized by means of functional arguments and polymorphic types, yielding such familiar tools as 'map', 'foldr' and 'filter'. A discussion of list-splitting functions provides more examples of generalization.

The only pitfall I see in this sequencing of topics is that it caters to many students' tendency to cling to the first methods they've mastered. For example, many students persist in defining functions recursively 'from scratch', and resist such powerful conveniences as list comprehensions and higher-order functions. No doubt some of the blame accrues to earlier exposure to such drudgery-intensive languages as Pascal and C. As Bird and Wadler have shown in their admirable textbook (Bird and Wadler, 1988), however, it is possible to introduce list comprehensions and higher-order functions *first*, delaying explicit recursion until one encounters problems in which it is really necessary. An instructor who prefers to delay recursion may be able to take Thompson's chapters out of order; if the text under consideration is the Miranda edition, the Bird and Wadler text would be a worthy alternative.

Part II continues with a chapter which revisits functions. Its focus on such topics as function composition, partial application, lambda expressions, and currying and uncurrying further widen the gulf separating Haskell from conventional languages. For students with Pascal and C backgrounds, the notion of functions as arguments and results of other functions takes a lot of getting used to, and the instructor may need to slow down through the curves.

The chapter includes a nice case study – a text-indexing program built as a composition of seven functions. Combining this indexer with the formatter of Part I makes a very worthwhile homework exercise, similar to one posed by Paul Abrahams (1976) (students enjoy learning that in those days it was part of an NYU qualifying exam). The chapter ends by extending the proof technique introduced in Part I to handle higher-order functions.

Up to this point, the two books are practically identical. The only significant differences correspond to differences between the languages' syntaxes, and to Haskell's distinction between integers and floats. From here on, however, the books diverge somewhat, as Haskell's type classes come into play.

Classes are motivated well, as a means of handling function overloading in Haskell's polymorphic type system, and the mechanisms for class and instance declarations are explained clearly. The differences between the Haskell and Gofer type systems are somewhat glossed over, but a full-bore treatment would be out of place in this introductory text.

Type checking gets a chapter of its own. Since types play such a central role in Haskell programming, this is a chapter well spent. The requirements for type correctness are presented as a set of rules, which accords nicely with the book's emphasis on calculation, but the discussion is a bit terse, and seems to look at type checking more from the point of view of the implementor than of the novice functional programmer. Students need reminding that type checking is not an arbitrary hurdle placed in the programmer's way, but a most valuable service (getting the types right is most of the battle). They also need some help in interpreting the type checker's diagnostic messages; this chapter's rule-based presentation is a good start, but a few more examples (just a pointer to the appendix on error diagnostics would help), and quite a few more exercises, would round it out nicely.

Part III, 'Larger-Scale Programming', introduces algebraic types and abstract data types, with design guidelines and case studies, explaining and demonstrating their increasing importance as software projects scale up. Also included are chapters on lazy evaluation, input/output, and computational complexity. An entire chapter is devoted to an extended case study on Huffman codes. A case study on simulation begins in the chapter on algebraic types and ends in the one on abstract types, providing a nice thread of continuity.

Particularly in this last third of the book, not only is there far more material than could be covered in a single semester, but much of the material is probably beyond the reach (not to mention the aims) of most beginning students. This observation is not meant as a criticism; the extra material (e.g. lazy calculation rules for list comprehensions, proofs involving infinite lists, monads, space complexity, and memoization) makes the book useful for upper-level functional-programming courses, and many beginning students will want to keep it in their personal libraries for further study – even if they never take another course in functional programming.

For a first course, the most essential elements of Part III would be algebraic types (because they are used heavily in implementing ADTs) and abstract data types (because they are used so universally in modern software development). In addition, exposure to the topics of the case studies – queues, binary search trees, sets, and graphs – in the abstract medium of Haskell prepares students for lower-level treatments of the same topics in subsequent courses using conventional imperative languages. I noted just one omission: Given the welcome emphasis on proofs throughout most of the text (including an extension to handle algebraic types), it is slightly surprising to find no discussion of interface specifications for abstract data types.

In addition to abstract data types, a beginning course needs to touch on input/output, including interaction, if only to dispel the possible impression that Haskell is limited to evaluating self-contained expressions (I generally introduce I/O much earlier, to improve Haskell's credibility in the eyes of students who are old hands at writing interactive programs in Pascal or C). The Haskell book offers sections on both stream I/O and monadic I/O; in my introductory course, I settle for the former, assuming that at this level monadic I/O would give rise to more confusion than it would relieve (but this book's explanation of monads is one of the better ones, and I plan to work it into my programming-languages course).

Laziness could be omitted from a beginning course on the grounds that it's of concern only to functional-programming specialists. In my course, however, I make sure to include Thompson's case study on parsing expressions, which follows Wadler's (1985) list-of-successes approach. Not only does this section impart, with amazing conciseness, a high-level understanding of parsing which will serve many students well in later courses, it's a convincing tour de force for recruiting functional programmers. Building and running the interactivecalculator case study, and adding their own enhancements, has proved to be an exciting final homework project for my students.

Finally, an introductory course might well include at least part of the final chapter, on computational complexity. The material on functions' growth rates and time-complexity of algorithms is independent of programming paradigm, and prepares students for more rigorous treatment in subsequent courses.

The book's back matter includes eight appendices, a plump bibliography, and a good index containing pointers to many (though not all) of the functions introduced in the text. Some of the appendices provide routine reference material – a useful glossary, suggestions for further reading, a list of Haskell's operators, a discussion of Haskell and Gofer error messages, and pointers to sources of Haskell implementations.

One appendix is devoted to a comparison of functional programming with imperative programming. There seems to be an implicit assumption that after completing this book on functional programming, students will be moving on to their first encounter with imperative programming. At least in the US, however, a majority of CS students have a year or two of pre-university programming, invariably (and regrettably) imperative. The book could help these students through the transition from imperative thinking to functional thinking by interspersing the comparisons in the main part of the text, pointing out both the similarities and the differences as they arise. The disruption, for students who have no imperative habits to overcome, could be minimized by presenting the comparisons in sidebars.

All told, however, each of these books is a major contribution to the practicality of teaching functional programming using Haskell or Miranda. An instructor setting up a functional programming course, for either novice programmers or upper-level undergraduates, can adopt either of these books in full confidence that it hits all the essential topics, and that it contains more than enough high-quality material for the most intensive course.

References

Abrahams, P. W. (1976) On realism in programming examples. *SIGPLAN Notices*, February.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need lambda calculus. *22nd ACM Symposium on Principles of Programming Languages*. San Francisco, CA.

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice-Hall.

Wadler, P. (1985) How to replace failure by a list of successes. *Functional Languages and Computer Architecture* Nancy, France. (*Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.)

Hamilton Richards Jr.

Higher Order Operational Techniques in Semantics edited by Andrew D. Gordon and Andrew M. Pitts, Cambridge University Press 1998, ISBN 0 521 63168 8 (hardback).

This book consists of a collection of articles on the subject by various established authors. Its origin goes back to a workshop of the same name that took place in October 1995. From this fact it is not difficult to see why the format resembles conference proceedings: the articles are thematically connected but independent from each other. Still, each article on its own is longer and more detailed than any conference proceedings would permit – in other words, what we have here is a halfway house between a proper text book and conference proceedings. This is probably the best one can do at the moment, the subject does not seem to be quite ready yet for a text book.

This is a book about semantics. Semantics without any strings attached – no lattice theory, no game semantics, no domain theory, no fancy stuff at all. We are talking operational semantics here: the meaning of a program is what it can do, not what it 'is'. To put it in another way: the book is about the kind of semantics that appeals more to programmers than to mathematicians, partly because it reflects their way of thinking, partly because writing an operational semantics is very similar to writing a compiler. One may argue whether this is an advantage but for certain programming languages (or programming language constructs) it is the only semantics we have anyway.

What *is* operational semantics? It used to have a bad reputation (when compared to denotational semantics), and some people might have argued then that this question has no precise, mathematical answer. But it probably has: we give an operational semantics by defining a coalgebra of some endofunctor – for programs without I/O we could use the category of sets and the powerset functor. Less abstractly speaking, the coalgebra has the programs as inhabitants of the carrier sets and their potential behaviours as coalgebra structure; in the most restrictive form (powerset functor) we only get the distinction between doing anything and doing nothing.

Dually to an algebraic semantics, where the object of special interest is the initial algebra – we keep everything distinct that has not been specified as equal, the coalgebraic semantics singles out the terminal coalgebra, where we identify everything we cannot distinguish. I should perhaps emphasize that this is *my* explanation, not the subject of the book. The book is more concerned with concrete operational semantics (particular coalgebras if you like), thus the definition of programs and their execution behaviour.

Why write a book about it? It is much easier to define an operational semantics than to reason about it. The reason we can reason at all (reasonably) about operational semantics is that it is structural: our reduction relation is given inductively, by structural induction on programs. However, the reasoning about programs is coinductive, not inductive, just as the semantics is coalgebraic and not algebraic, since equality in this world is derived and inequality primitive: two programs are equal if we cannot distinguish them.

This is the place where 'higher-order' enters the play: we establish equality by finding an appropriate bisimulation. A bisimulation is a relationship between the states of two labelled transition systems (or we could say: coalgebras) that shows them to behave identically; it is the coalgebraic dual of the algebraic concept of a congruence relation. The reason we talk about relations in the plural is that bisimilarity (of programs) is not even semi-decidable. So one shows it in practice by constructing a specific bisimulation relation for the problem at hand. Thus we have to reason *with and about* relations.

Everything you always wanted to know about this can already be found in Gordon's article on object calculi, as well as the things that explain why you were afraid to ask about it in the first place: a mind-boggling collection of subtly distinct preorders and equivalences may startle the uninitiated. As one goes through the book one is often struck by a feeling of deja vu: concepts and terminology and proof techniques reappear in slightly different guises (for different programming calculi) and it is a bit of a shame that there is no recognised common abstract theory in this field that liberates one from the syntax.

Not every article in the book is occupied by such fundamental problems: several articles have more concrete concerns. For example, Morrisett and Harper address the problem: is our garbage collector correct? They show how operational semantics provides a vehicle to pose this question meaningfully (no mean feat) and how the techniques established elsewhere help to settle the question. In the positive, of course – the good guys always win.

To sum up: it is an interesting book. Much of the material can be found elsewhere, but not in one volume and rarely up to this degree of detail. I would expect over time a common theory of operational semantics to emerge which would supersede much of the contents of the book. But that's the future. If you want to know something about operational semantics *now*, start here.

Stefan Kahrs