Book reviews

Programming with Standard ML by Colin Myers, Chris Clack and Ellen Poon, Prentice Hall International, Inc., New Jersey, 301pp, 1993, ISBN 0-13-722075-8.

ML for the Working Programmer by L.C. Paulson, Cambridge University Press, 1991, 429pp, ISBN 0-521-39022-2.

Elements of ML Programming by Jeffrey D. Ullman, Prentice Hall International, Inc., New Jersey, 1994, 320pp, ISBN 0-13-288788-6, 0-13-184854-2 (USA).

Introduction

The ML family of languages has been a hugely influential vehicle for the development of a number of programming language features (Milner, 1983). *Standard ML,* defined in Milner *et al.* (1990), has emerged as a mature language, SML, with several efficient implementations: *New Jersey ML, PolyML, Poplog ML, MLWorks.* This language incorporates many 'advanced' features: first-class functions, recursive types, abstract types, polymorphic type-inference, exceptions, references, modules, pattern-matching. SML is not a small language; it is surprisingly hard to teach (but not so hard as C). Nevertheless, it has grown popular as a teaching language, and, to a much more limited extent, for the implementation of production code.

The first ML textbook, by Wikström, *Functional Programming using Standard ML* (1987), was designed to teach programming to novices. The growing popularity of SML has stimulated many more authors to write ML-based text books with a pragmatic orientation. Typically, these later books assume some knowledge of programming, and use ML to introduce a more structured approach. "In contrast to the rather idealistic books [on functional programming] that are the norm" (as Paulson puts it), they seek to develop practical programming skills.

This review compares and contrasts three such books: Paulson, *ML for the Working Programmer* (1991); Ullman, *Elements of ML Programming* (1994); and Myers *et al.*, *Programming with Standard ML* (1993) – the bibliography contains several more. How do the authors approach this task? How do these texts compare with Wikström? How are the books suited to the student, or the working programmer? How can they be used in teaching?

Summary

Of these three, Paulson is my personal favourite, but Ullman will be more appropriate for some purposes. Paulson's programs are a pragmatic blend of elegance and efficiency, with well-chosen data-structures and algorithms. The book provides a wealth of examples of carefully crafted code – paradigms of sound software design. As a reference, Ullman has the most accessible organisation, the best index and outstanding coverage of common pitfalls.

This is the book for the programmer wanting a self-contained introduction to ML, or the teacher wanting to cite an accessible reference text.

Paulson could be used as a template for a number of innovative courses, but it is of limited value as a reference unless the lectures follow Paulson's development fairly closely. Only the more able students will find that Paulson gives an easy introduction to ML; for the rest Ullman, or Myers *et al.*, give a more gentle introduction. Ullman could serve as a generic text to under-pin a wide variety of ML-based courses. Harper's technical report, *Introduction to Standard ML* (1986), provides a somewhat terser, but still accessible, introduction. Neither Ullman nor Harper is appropriate for teaching ML as a *first* programming language; for this, Wikström (1987) is still the best available.

Paulson and Ullman provide syntax charts for ML. All three books include lists of predefined functions (Paulson's is on the end-papers; many readers never find it). Ullman is based on *New Jersey ML*, which has significant variations from the standard. The other books (and compilers) are more faithful to Milner *et al.* (1990). Wikström also provides an easy reference, but it was written before the final version of the standard (Milner *et al.* , 1990), and has not been updated (the changes to the 'Core' language are minimal, but there is nothing on Modules – which came later).

The treatment of modules is, in general, the weakest part of these books. Effective use of modules requires a conceptual jump: when programming 'in the large', a well-designed, and efficiently implemented, abstract data-type is a beginning, rather than an end. Paulson makes this leap, but in the process he may leave some readers behind. The other two develop the application of structures and signatures to encapsulation, but only scratch the surface of the use of functors in parameterisation.

The module system is subtle; thoughtful readers of these books will be left with unanswered questions. For a lucid exposition of (some of) the subtleties, see Tofte's brief report *Four Lectures on Standard ML* (1989).

Myers, Clack and Poon

This is a book *about* structured programming; its aim is to use ML to develop sound software design and code-management techniques. The text works through examples, often starting with a flawed design which is then improved. A pattern-matching program – loosely based on UNIX *grep* – is taken as an extended example. There are many exercises, most with solutions.

The book has seven chapters: Operators, Values and Types; Functions; Lists; Curried and Higher-Order Functions; Managing Environments; User-defined Types; Modules. The *grep* example is an excellent idea: the patterns matched are extended from literal strings to include character ranges and repetitions. Although the authors call the final patterns 'regular expressions', their representation of patterns does not provide for nested concatenation and repetition – so, for them, $(ab)^*$ is not a 'regular expression'! This is unfortunate, both because the terminology may confuse students who go on to study computer science, and because regular expressions would have provided a good example of a recursive data-type.

I found this book difficult. The style, 'which shows both correct and incorrect approaches to problem solving', is appropriate for a tutorial – where individual misconceptions may be dealt with as they arise – but problematic in a textbook. There is much good advice, but the reader has to work hard to disentangle the good from the bad; the incorrect approaches sometimes obscure the essential simplicity of the examples.

Ullman

This is a book *about* ML, intended "as a tutorial and reference for the person who wants to program productively in ML". Ullman takes a straightforward approach: he uses examples to motivate the various features of the language, and points out common pitfalls. The examples and exercises also cover a number of data-structures: notably, binary search trees and 2-3 trees.

The text is organised in three parts. The first is an introduction to programming in ML: basic functional programming, expressions, lists, function declarations using patterns, `let`, exceptions and I/O. The second part covers 'advanced' features: polymorphic and higher-order functions, datatype declarations, modules, arrays, references. Part three summarises the syntax and built-in functions of ML in some detail, and covers additional features: records, `case` expressions, exception handling, curried functions.

There is a clear progression in the introduction of new features, and a well-considered index. Potential misconceptions are identified with 'bullets', and clearly separated from the main train of thought. Ullman's bullets alert the reader to particular dangers and pitfalls, providing the novice with the benefits of experience. Ullman tends to shy away from subtle issues, and sometimes avoids potential confusions by telling only part of the story (for example, on imperative type variables, and type identity) – effective teaching often depends upon such judicious omissions.

As a tutorial and reference for the ML language, Ullman is excellent. This masterly textbook comes from an author with a proven track-record. But mastery of Ullman's book will not produce a master-programmer; practical use of ML requires more consideration of complexity, and informal knowledge of some basic tricks of program transformation – for example, the introduction of accumulating parameters.

Paulson

Paulson also aims to provide a guide to the effective use of ML, but this book is *about* programming. Paulson *is* a working programmer. He has long experience of using various dialects of ML, and contributed to the development of the standard. His book is intellectually stimulating; he delights in seizing an idea and worrying at it, like a cat toying with a mouse. Many students find this off-putting – perhaps they identify too closely with the mouse.

The first chapter discusses functional programming at an abstract level, using mathematical examples which may already alienate some readers. Chapter 2 introduces expressions, functions and (polymorphic) types – the pace is brisk, and invigorating. Chapter 3 deals with lists. These are applied to represent matrices (Gaussian elimination is developed as an example), sets, association lists and graphs. Sorting is discussed briefly but in depth. Trees and datatypes come next, in Chapter 4. Again the pace is hectic: datatypes, enumeration types, exceptions; trees and tree traversal, binary search trees, priority queues (an idiosyncratic treatment), and abstract syntax trees (a tautology checker as an example). Chapter 5 covers functions as values: higher-order functionals, streams (infinite lists), and search strategies. Chapter 6, on reasoning about functional programs, is central to Paulson's approach: "Correctness must come first. Clarity must usually come second, and efficiency third. [...] A judicious mixture of realism and principle, with plenty of patience, makes for efficient programs." Modules are covered in Chapter 7, which we discuss in detail below. Chapter 8 is devoted to imperative features of ML: references, arrays, I/O. Paulson *applies* references to build ring buffers, lazy streams and version arrays. He builds pretty-printing tools. Chapter 9 applies ML, in earnest, to build a λ-calculus interpreter. He introduces a beautiful set of (top-down) parsing tools (following Burge, 1975), de Bruin indices, call-by-name and call-by-value evaluation, and the Y combinator. Chapter 10 concludes the book, with a tactical theorem prover for first-order logic – including unification, parsing and pretty-printing.

Some details

To give a flavour of the three approaches we contrast their treatment of two topics, the alpha and the omega of ML programming: list reversal, and modules.

List reversal

Although this is provided as a basic function, rev, in the standard *Initial Basis*, all three authors use it as an example of recursion over lists.

- Myers *et al.* introduce list reversal in Chapter 3 as an example of 'case analysis'. Enumerating combinations of patterns and expressions that might produce type-correct clauses produces the clauses we want, together with others, such as
  ```
  reverse (front :: rest) = [hd rest, front]
  ```
  After rejecting such monstrosities, a first 'correct' implementation has four clauses; lists with one or two elements are treated as special cases. Once these redundant cases are removed we have the standard $O(n^2)$ implementation in terms of append. Although 'accumulative recursion' has already been discussed (in Chapter 2), the $O(n)$ version is only mentioned *much* later. Even then, complexity is not mentioned.
- Ullman uses naïve list reversal as a first example of recursion. He makes two ideas fundamental: a basis, and an induction step with a recursive call for a 'smaller' argument. His next example is binomial coefficients ${}^nC_m$, to illustrate non-linear recursion. Accumulating parameters are *never* mentioned.
- Paulson introduces recursion, iteration and accumulating parameters using the factorial function as an example. The complexity of both versions of reverse is discussed in the chapter on Lists. In Chapter 6 structural induction is applied to prove they are equivalent. What a shame that the prover of Chapter 10 is not developed to the point where this theorem can be proved!

Modules

SML has safe abstract types and a module system. The module system introduces many new ideas: structures, signatures, functors and sharing. It is subtle; none of these books gives a complete and uncluttered account. It is also more powerful than the single level of functor application described by Ullman and Myers *et al.* might suggest. Both omit any example of sharing between functor parameters. The application of modules to a system with a non-trivial module dependency graph depends crucially on such sharing constraints.

- Myers *et al.* introduce abstract types in Chapter 6, using natural numbers as a first example. In Chapter 7, all the components of the module system are described. Code for the ongoing pattern-matching example is parameterised on a lexical analyser. ML modules are used simply, but to good effect, to separate the lexing of patterns from the pattern matching code, and give a convincing example of code reuse.
- Ullman devotes two chapters to modules, one on the basic constructs of the module system, and one, 'Software Design Using Modules', on information hiding and abstraction; this covers the New Jersey specific `abstraction` construct, as well as the standard abstract types, and the use of signatures to hide information. In introducing functors, using, as an example, a search tree parameterised on a type with an ordering, he lapses into the 'correct and incorrect approaches' style; I prefer his usual, direct and correct, style.
- Paulson applies the module system to build, for example, priority queues from arrays, which, in turn, are built from trees. He discusses the system, and its shortcomings, in some detail. He eschews the "15-year-old [abstype] construct [which] ought to be superseded by modules" (but is not, except in the New Jersey `abstraction` extension – abstract types provide the most straightforward secure type abstraction in *Standard* ML). Paulson's parsers and pretty-printers are packaged as functors. The final chapters demonstrate, for those who read this far, the effective use of ML modules to construct significant software systems from independent reusable components.

References

MLWorks. (compiler), Harlequin Ltd., Cambridge, England. Contact `works@harlqn.co.uk`.

PolyML. (compiler), Abstract Hardware Ltd., Uxbridge, England. Contact `ahl@ahl.co.uk`.

Poplog Standard ML. (compiler), Integral Solutions Ltd., Basingstoke, England. Contact `isl@integ.uucp`.

Standard ML of New Jersey. (compiler), Bell Labs, Priceton, New Jersey. Available from `ftp.research.att.com` by anonymous `ftp`.

Burge, W. H. (1975) *Recursive programming techniques.* Systems Programming Series. Addison-Wesley.

Harper, R. (1986) Introduction to Standard ML. *Technical report* ECS-LFCS-86-14. LFCS, University of Edinburgh, Department of Computer Science.

Milner, R. (1983) How ML evolved. *Polymorphism (the ml/lcf/hope newsletter)*, **1**(1).

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

Myers, C., Clack, C. and Poon, E. (1993) *Programming with Standard ML.* Prentice Hall.

Paulson, L. C. (1991) *ML for the Working Programmer.* Cambridge University Press.

Reade, C. (1989) *Elements of Functional Programming.* Addison-Wesley.

Sokolowski, S. (1991) *Applicative Higher-order Programming: The Standard ML Perspective.* Chapman & Hall.

Stansifer, R. (1992) *ML Primer.* Prentice Hall.

Tofte, M. (1989) Four Lectures on Standard ML. *Technical report* ECS-LFCS-89-73. LFCS, University of Edinburgh, Department of Computer Science.

Ullman, J. D. (1994) *Elements of ML Programming.* Prentice Hall.

Wikström, Å. (1987) *Functional Programming using Standard ML.* Prentice Hall.

MICHAEL FOURMAN
Abstract Hardware Ltd., 1 Brunel Science Park,
Kingston Lane, Uxbridge UB8 3PQ

Laboratory for Foundations of Computer Science, University of Edinburgh,
King's Buildings, Mayfield Road, Edinburgh EH9 3JZ