Book review

Abstract Data Types in Standard ML by Rachel Harrison, John Wiley & Sons, 1993, 212 pp, ISBN 0-471-93844-0.

The aim of this text is to present a thorough treatment of data abstraction within a functional framework. Although there is now an abundance of introductory books on functional programming, this is one of the few to be aimed at a second course, and the first to be aimed specifically at the data structures course commonly taught in universities. The philosophy of the text is the same as Harrison's earlier book on Modula-2, and is similar in style to many other second course imperative programming texts. The main novelty stems from the use of Standard ML (SML) as the implementation language. SML is ideal for this purpose due to the explicit support for abstract data types (ADTs) provided by the **abstype** and module constructs of the language.

After an initial chapter outlining the philosophy behind the use of abstract data types, and a brief introduction to functional languages, the book devotes one chapter in turn to each major type of ADT: lists, stacks, queues, and so on. An axiomatic specification is used to define each type, followed by the development of an implementation and examples of its use. Each chapter concludes with a summary reviewing the main definitions and concepts, which is very helpful, and acts as a quick revision aid. Functions are specified informally throughout the book using pre- and post-conditions.

Chapter 2 deals with lists, and covers most of the elementary list processing functions, together with a fairly comprehensive treatment of sorting. The chapter also describes how to give a representation independent specification of an ADT using a set of axioms. Consistency and sufficient completeness of the axioms are discussed, and an informal proof of these properties given for the list axioms. The corresponding proofs are omitted for all the other types in the book. Chapter 3 covers the stack ADT and the initial implementation is based on SML's **abstype** construct. Deficiencies in this implementation motivate the need for functors and structures, and an implementation using these is then developed. Chapter 4 presents queues as an ADT, first using a list as the underlying implementation type, and then using a pair of lists for increased efficiency. Priority queues are also treated, and dequeues discussed but no implementation given. Chapter 5 deals with binary trees and then Chapter 6 extends this material to ordered binary trees. The sorted binary tree is used as the basis of an implementation of a priority queue. Deficiencies in this approach then motivate the material on heaps and heapsort. Finally, the chapter discusses balanced trees, and in particular AVL trees, although no implementation is provided. Chapter 7 is devoted to a lengthy discussion of 2-3 trees and their implementation. Chapter 8 deals with finite sets and multisets, implemented as lists and ordered trees. The final chapter deals with graphs, and graph algorithms such as graph searching and topological sorting.

The book includes exercises, typically involving the construction of a short function, or the evaluation of an expression by hand. They are ideal as quick exercises for reinforcing the material in the book. From a teaching perspective it might have been useful to include some more difficult exercises that could form the basis of a project or practical. No solutions to the exercises are provided.

When programming in SML there is frequently some tension between the use of concrete

data types, with their associated support for pattern matching, and abstract data types supporting representation independence. Lists and trees would usually be represented as concrete types, whereas types that were not freely generated, such as sorted trees and sets, would be modelled as abstract types. The book is rather inconsistent when it comes to the treatment of this subject. For example, lists are first introduced as an ADT, but then nearly all the examples are based on pattern matching, treating lists as a concrete data type. Trees, on the other hand, are viewed purely as an ADT. It is unclear why lists and trees should be distinguished in this way, and such differences may confuse a reader unfamiliar with programming in SML. This is perhaps an example of where the traditional approach to ADTs has not been modified to take into account the facilities and advantages of the language being used.

Most of the abstract types in the book are implemented by functors parameterised on a structure called `Item`. This structure contains details of the object being manipulated by the ADT, such as its type, an equality function, an ordering, and a print function. The approach is appealing as one would expect that the same structures could be used to build many different container types. An `Item` structure for integers could be used to build stacks of integers, queues of integers and sets of integers for example. Unfortunately, in the book each ADT requires its *own* specialised `Item` structure. The `Item` parameter for stacks must include a function to print out stacks and the one for queues prints out queues and so on. It is not at all clear why these functions were not included in the functor, allowing the same structures to be used in a variety of contexts. One of the aims of a volume on ADTs should be to develop a sound methodology for creating and using such types, and this book seems flawed in this respect.

Deciding which functions to provide within an ADT, and which to define outside its barriers, is a problem frequently encountered by students when they start to define ADTs. There is often a trade-off between efficiency and the effort required to alter the representation type. The author has made some reasonable choices for most of the types, though it would have been helpful to have explained the rationale behind these choices more explicitly. The interface to the Set ADT is a little more *ad hoc*, providing set subtraction and the ability to remove an element from the set, but no mechanism for iterating over the elements of the set for example.

The book contains a number of programming errors that might confuse a novice programmer. To cite a particular example, in a number of places an **open** declaration is used in a functor body with the unintentional side-effect of hiding some of the functor parameters. The error is compounded by the omission of the sharing constraints that would otherwise have been required. In other examples, where an **open** declaration has not been used, the sharing constraints have been included. The language features used in the book are only described very briefly, and so it may be difficult for a reader to understand such problems without external assistance.

Such deficiencies would be easy to fix, either in a revised version of the book, or by a lecturer basing a course on this text. A more worrying problem concerns the efficiency of the example code presented in the book. Functional languages are often perceived to be inefficient, both in terms of execution speed and memory usage. The author argues that this is becoming less of a problem due to improved compiler technology. The book emphasises efficiency concerns by noting the time complexity, though not the space complexity, of many of the functions. Unfortunately, some of the functions seem to be written with a fairly reckless disregard for efficiency. It is not uncommon to find code such as

```
if height l > height r then ...
else if height r > height l then ...
else ...
```

where potentially costly expressions are executed repeatedly. In a textbook there is, of course, a case for emphasising clarity rather than efficiency. However, many of the functions are

neither clear or efficient. For example, to check if a list is sorted the author takes the list and pairs it up, element by element, with its tail. The pairs are then checked to ensure they are ordered. In a lazy language this might be acceptable as the intermediate list would only be constructed on demand. However, in a strict language such as ML the implementation needs to construct this entire list of pairs for each call of the function. There are many other similar cases in the book, leaving one with the impression that the code is not really suitable for serious use. The clarity of the code could also be improved by more careful typesetting of some of the larger examples.

The book is aimed at those readers who have some familiarity with a functional language, but not necessarily SML. For such readers the book should provide a good grounding in the principles of data abstraction within a functional framework. It is not clear whether the author also intended the book to be used by functional programmers wishing to learn SML. In my opinion, it would be difficult to use the book as a self-contained introduction to programming in SML for a number of reasons. Whole areas of the language are omitted (e.g. exceptions and references), whilst other parts are given only a cursory treatment (e.g. the material on modules). The quality of some of the code and programming styles might also give cause for concern in the context of teaching inexperienced programmers. The book is more successful if used as an advanced text when the reader has already developed a firm grounding in SML programming from reading a book such as Paulson's *ML for the Working Programmer*. In such cases, the initial sections of the book may be redundant, although some of the later chapters would be useful. The book could also be used as a reference manual of ADT implementations, although many of the examples would require further development before they could be considered as exemplary solutions. In summary, a commendable first effort in this area, and I look forward to others.

K. MITCHELL

Department of Computer Science

University of Edinburgh