



ELSEVIER

Science of Computer Programming 34 (1999) 75–77

Science of
Computer
Programming

Book review*

Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998, ISBN 0 521 63124 6, x+220 pages, hardback.

Purely functional programming languages have a reputation for being inefficient. Two reasons are often quoted: immature compiler technology and the difficulty of implementing classical abstract data types.

Advances in compiler technology have shown that functional languages can be compiled to competitive code c.f. the Pseudo-knot Benchmark. The main lesson here is that compiler technology that works in the imperative domain does not necessarily work in the functional domain.

Chris Okasaki shows that developing special techniques for standard algorithms can also solve the second problem in some important cases. *Purely Functional Data Structures* discusses familiar data structures in a functional setting. This includes single and double ended queues, binary search trees, heaps, tries and random access lists, all in a number of important varieties. Some familiar data structures are not covered in the book, like graphs. Hash tables are mentioned only briefly.

Purely Functional Data Structures is a revision of the authors Ph.D. thesis. It is neither a straight replacement for classical texts such as Sedgewick's Algorithms and Data Structures, nor does it claim to be such a replacement. The book is a little less accessible than I would have liked. I think that it could be used to supplement a classical text, which would provide a more gentle introduction to the main analysis techniques and to the big O-notation, and which would also cover topics not present in *Purely Functional Data Structures*.

The book has a table of contents, a good index and an extensive bibliography. It also contains exercises but unfortunately no answers are provided. At least a selection should have been made available via the Web. The book is quite theoretical in nature, but occasionally provides a 'Hint for the practitioner'. The first is presented on page 26, whilst discussing red-black trees:

Even without optimisation, this implementation of balanced binary search trees is one of the fastest around. With appropriate optimisations, such as Exercises 2.2 and 3.10 it really flies!

* Review copies of books which might be of interest to the readers of *Science of Computer Programming* should be sent to Prof. K. Apt (address: see inside front cover). Proceedings of conferences will not normally be reviewed.

The reader interested in actually using the algorithms and data structures will find the hints useful but lacking in depth. In the case of red-black trees for example a comparison with an imperative approach would have been illuminating: how much does it fly? Again such information could be provided on a Web site. The book consists of 12 chapters, which we will now discuss.

The first chapter introduces terminology, and discusses the differences between imperative and functional programming, as well as the distinction between eager and lazy evaluation (evaluate expressions only when needed, but do remember the results for possible re use later). The book presents its algorithms and data structures in SML (eager), introducing lazy evaluation through annotations where necessary. Haskell (lazy) versions of most algorithms and data structures can be found in the appendix.

The main property of data structures in a purely functional setting is their persistence. Chapter 2 discusses the standard abstract data types for persistent lists and binary search trees. Persistent implementations of these data structures in an imperative language would have precisely the same complexity as their functional counterparts. This chapter also seizes the opportunity to show off the elegance and power of the module mechanism of SML.

Chapter 3 continues the introduction of persistent data structures with leftist heaps, binomial heaps, and red-black trees. Okasaki shows that in a persistent setting the algorithms are simpler than in an ephemeral setting.

Amortisation is the key to developing purely functional data structures with a complexity that one would get in an imperative setting. This technique basically does a little extra work often, rather than a lot of work, all at once. In lazy functional languages, amortisation arises naturally, but in an eager language programming effort is required to obtain amortised behaviour. Chapter 4 describes an extension to SML, which makes lazy evaluation possible. Unfortunately the extension ($\$$ -notation) is not purely syntactic; it requires modifications to the compiler. There is also no discussion of the correctness of the extension, which is a weak point of the book. The alternative of using a lazy language would not have been entirely satisfactory either, because the analysis techniques whilst adequate for eager evaluation, have difficulty dealing with lazy evaluation. To make the algorithms and data structures more accessible to practitioners one needs an implementation of ML with the $\$$ -notation. Unfortunately, the extension provided by SML/NJ (`delay` and `force`) only goes some way towards supporting laziness.

Chapter 5 introduces the amortised analysis techniques used in the book. These are the bankers and physicists methods due to Tarjan and Sleator. Both methods are based on accounting for time by accumulating *savings* which can be spent later. The chapter presents the following data structures: the Batched queue (which is based on a pair of lists, and which needs the occasional reverse), double ended queues, splay heaps and pairing heaps. The data structures are analysed, comparing the worst case complexity with the amortised performance. As pointed out at the end of the chapter, the analysis techniques assume that the data structures are used in an ephemeral way. The data structures are in fact persistent, thus invalidating the amortised analysis technique.

Chapter 6 uses lazy evaluation to reconcile persistence with the amortised analysis techniques. The analysis techniques are now based on accumulating *debt* that has to be paid off. This helps because “savings can only be spent once, although it does no harm to pay off debt more than once” [page 59]. New versions of the data structures are presented, unfortunately without giving a correctness argument. The analysis results from the previous chapter for all but the Splay heaps are revisited. The latter “do not appear to be amenable to this technique” [page 59].

Chapter 7 introduces the idea of scheduling, which basically works as follows: Where the analysis technique would just accumulate debt, the algorithms actually do some work, corresponding to paying this debt off regularly. In this way, an amortised data structure becomes a worst case data structure. The implementation of Batched queues now requires two lazy lists and an eager list to provide a handle on spreading the cost of the list reverse evenly. The implementation is a true real-time queue. It is reasonably complicated but still quite readable, in spite of the heavy use of the $\$$ -notation. The same cannot be said for scheduled binomial heaps, which are now quite complicated.

Chapter 8 discusses lazy rebuilding, which basically achieves the results of scheduling but without relying on laziness. The algorithms are considerably more complicated. The reason is that the state of various operations has to be encoded in the data structure, and subsequent operations must take the various states of previous operations into account.

Chapter 9 exploits the structural relationship between number systems and data structures. For example a list is closely related to the unary number system. Operations on numbers such as addition then correspond to changes of the data structure. The results include a random access list with $O(\log n)$ worst case performance for lookup and update and $O(1)$ for all other operations.

Chapters 10 and 11 differ from the previous chapters in that they describe various related techniques that allow new data structures to be derived from old data structures. The old data structures are either less efficient or, incomplete. To fully benefit from data structural bootstrapping, SML should have had a more powerful type system (based on non-uniform recursion). In this case Okasaki shows how one can get by without a special version of SML.

Summarising, *Purely Functional Data Structures* is an important step towards the development of the theory purely functional data structures. The book is unfortunately less accessible to the practically minded.

Pieter Hartel
University of Southampton