The Monad.Reader/Issue3/Purely Functional Data Structures

From HaskellWiki

< The Monad.Reader | Issue3

Andrew Cooke (http://acooke.org/) 's review of **Purely Functional Data Structures** was originally published on Slashdot (http://books.slashdot.org/article.pl?sid=04/02/19/2257203&tid=156&tid=192&tid=198&tid=6) and is reprinted here with permission from the author.

Contents

1 Review: Purely Functional Data Structures

A while ago I read the comments following a Slashdot book review. Someone had posted a request for books that covered a wider range of languages than Java, C, Python, etc. Well, I thought, why not review Okasaki's Purely Functional Data Structures? It's a classic from the underworld of functional programming - recognised as the standard reference, yet clear enough to work as an introduction to the subject for anyone with a basic functional programming background. Of course, some readers won't know what functional programming is, or what is special about pure data structures. So I hope that this review can also serve as something of an introduction to the languages that I (a software engineer paid to work with Java, C, Python, etc) choose to use in my spare time, just for the joy of coding." Read on for the rest; even if you're not planning to give up C or Perl, there are links here worth exploring.

- *Purely Functional Data Structures*

  author: Chris Okasaki
  pages: 220
  publisher: Cambridge University Press
  rating: 8/10
  reviewer: Andrew Cooke

ISBN: 0521663504
summary: Functional programming for grown-ups.

1.1 Introduction

In Okasaki's introduction he says that the "[...] benefits of functional languages are well known, but still the vast majority of programs are written in imperative languages such as C. This apparent contradiction is easily explained by the fact that functional languages have historically been slower than their more traditional cousins, but this gap is narrowing."

Indeed, OCaml (http://caml.inria.fr/) has a reputation for being "as fast as C," yet it contains automatic memory management and supports object-oriented, as well as functional, programming. It's also probably the most widely used functional language outside academia (except perhaps Lisp/Scheme).

I mention OCaml not just because it's fast, free and popular, but because Okasaki uses a related language - ML - in his book. The ML family of languages are the standard strict, functional languages (Standard ML of New Jersey (http://www.smlnj.org/) is perhaps the reference implementation but see also standardml.org (http://www.standardml.org/)). Okasaki also includes an appendix with examples in Haskell, which is the standard lazy functional language.

1.2 Laziness

The difference between lazy and strict languages is the order in which code is evaluated. Most languages are strict. Unlike most languages, Haskell only evaluates something when it is absolutely necessary. Each parameter to a function, for example, is passed as a "thunk" of code, not a value. If the value is not required inside the function, the parameter is left unused; if it is required (say as part of a result that needs to be displayed) then the thunk is evaluated. This evaluation may trigger a whole slew of evaluations of functions that "should" have been called earlier (from a Java programmer's point of view).

Laziness is both good and bad. The bad side is obvious: the order in which code is executed my be very different from the order in which the program was written and some serious book-keeping is necessary in the compiler to juggle the thunks of code and their final values. The reordering of code could cause mayhem for IO operations, for example (in practice, of course, Haskell includes a solution to this problem).

The good side is that laziness can help make programs more efficient and, while the definition of ML doesn't include laziness, individual ML implementations - including OCaml and SML/NJ - include it as an extra.

Much of *Purely Functional Data Structures* (the second of three parts) focuses on how to use laziness to make data structures efficient. Lazy evaluation allows book-keeping actions to be postponed, for example, so that the cost of maintaining the data structure in an efficient form can be averaged across several read/write operations (improving worst case limits - avoiding a very slow response if the data happen to be in a "bad" order).

An understanding of how the efficiency of algorithms is rated (the big-O notation) is one piece of knowledge that this book does assume, along with a basic grasp of what Stacks, Queues, Trees, etc, are.

This lazy boost in efficiency is needed because, even though functional languages may be getting faster, it's not always possible for them to implement the efficient algorithms used in imperative (non-functional) programming.

1.3 Functional languages

But I'm getting ahead of myself, because I haven't described what a functional language is, or why it is useful. These are the topics of the first part of the book, which explains how functional languages, which make it impossible to change variable values by direct assignment, support persistent data structures. This section is fairly brief, and while it's a good refresher course for someone who's not had to worry about such things since studying at university, it's not sufficient as an initial introduction to functional programming in general.

There's a good explanation of functional programming (http://en.wikipedia.org /wiki/Functional_programming) in the Wikipedia (see also Functional programming), but, in all honesty, I don't see how anyone can really "get it" without writing functional code (just as I, at least, couldn't understand how OOP worked until I wrote code that used objects).

So forgive me for not telling you why functional programming is good (This paper is one famous attempt (http://www.md.chalmers.se/~rjmh/Papers/whyfp.html)), but perhaps a better question to focus on is "Why should you spend the time to investigate this?" The best answer I can give is that it leads to a whole new way of thinking about programming. Describing functional programming as "excluding assignment to variables" doesn't do justice to the consequences of such a profound change (one I found almost unimaginable - how can you program without loop counters, for example?).

There's a practical side to all this too - learning new ways of thinking about programs makes you a better programmer. This ties in closely with the final part of Okasaki's book, which explores a few fairly esoteric approaches to data structures. Who would have thought that you can design data structures that parallel the way in which you represent numbers? Some of this is pretty heavy going - I can't say I understood it all, but I'm taking this book with me on holiday (it's slim - just over 200 pages) and I'll be bending my brain around some of the points in the last few chapters as I lie in the sun (here in the southern hemisphere it's late summer).

So just who would benefit from this book? It seems to me that it's most valuable as a second book on functional programming. There are a bunch of texts (and online guides) that can get you started in functional programming. This one goes further. It shows how to exploit the features of functional languages to solve real problems in applied computing. Admittedly, they are problems that have already been solved in imperative languages, but you might find that you, too, come to enjoy those famous benefits of functional languages. The algorithms in this book let you enjoy those benefits without paying the price of inefficiency.