JFP **25** (e2): 1–3* (First pubd online 2015) doi:10.1017/S0956796815000076
*Provisional – final page numbers to be inserted when paper edition is published

Book review

Thinking Functionally with Haskell by Richard Bird, Cambridge University Press, 2014.
doi:10.1017/CBO9781316092415

With *Thinking Functionally in Haskell* Richard Bird steps up to continue a family of textbook classics. Bird and Wadler jointly started the series with two editions of *Introduction to Functional Programming (in Haskell)* in 1988 and 1998, respectively. Let me begin with the outright spoiler that I think that this third edition breathes new life into the series and indeed presents a worthy continuation.

The twelve chapters of the 340-page volume contain more material than most layouts of a one-term introductory course on functional programming can accommodate. If the many exercises are considered in depth and a discussion of Haskell-specifics is added (more on both points below), we hold the syllabus of a two-term course in our hands. According to the blurb, the book addresses first- or second-year undergraduates. I agree, but gained the impression that true programming novices would probably struggle starting with the fifth chapter when concepts, scripts, and exercises become more complex. This is also where the exposition generally picks up speed.

From the outset, Bird consistently adopts a style of programming in which complex functions are composed from simpler constituents that are useful on their own ("*. . . functions that seem to be basic in programming are often composed of even simpler functions. A bit like protons and quarks.*") Already the earliest exercises in Chapter 1 adhere to this principle: an elaborate pipeline of function types has to be designed even before students can be expected to write the functions' bodies. Early on lazy evaluation is established as a principle that makes this rigorous compositional style viable. Here, and at many occasions later in the book, Bird relates the discussion to the research literature or blog posts.[1] This provides welcome entry points for deep dives into the subject. When Chapter 4 introduces lists it carefully distinguishes finite, infinite, and partial values, a discussion that has its dedicated Chapter 9 but permeates the entire book.

Chapter 5 is entirely devoted to a Sudoku solver that readers of Bird's *Pearls of Functional Algorithm Design* (Cambridge University Press, 2010) will recognize. The present book significantly expands on the earlier treatment through an in-depth discussion of the many involved component functions. The derivation of an efficient solver from the "*clearest specification*" also marks the first larger showcase of equational reasoning. Bird consequently uses the "Wim Feijen style"

$$\begin{array}{ll} & e_1 \\ = & \{\mathit{justification}\} \\ & e_2 \end{array}$$

to simplify and optimize programs or to establish proofs of their properties. The rewriting of compositions of functions remains one of Bird's grand themes. Calculations pervade the entire book,

[1] Regarding a discussion of strict vs. lazy evaluation, Bird points to a blog post by Robert Harper and the extensive thread of comments that followed (`http://existentialtype.wordpress.com/2011/04/24`).

from its preface(!) to the final Chapter 12 where a (semi-)automatic equational rewriter is developed. Lawful program construction encourages "*wholemeal programming*", "*prevent[s] a disease called indexitis*", and explains "*why functional programming is the best thing since sliced bread*." Richard Bird is not shy to advertise the style and consistently lives by it. Chapter 6 on the induction over natural numbers and (partial and infinite) lists consequently reads like a particular exercise in equational reasoning.

A lazy language promotes the compositional approach but makes it harder to assess program performance. Chapter 7 addresses efficiency and provides an accessible introduction to issues like common subexpression elimination and space leaks. Here is where Bird introduces Haskell's `seq` primitive, "*the eager button on our dashboard*"—one of the few places where Haskell-specifics find their way into the text. A simpler eager evaluation model is then also used in an asymptotic analysis of program run time, a reasonable trade-off to make in a textbook.

Library design, and domain-specific languages in general, are the focus of Chapter 8. I felt that the chosen pretty printer showcase turned out to be so intricate, however, that genuine matters of library and DSL design were obscured. A related comment applies to the already mentioned final chapter: the development and subsequent optimization[2] of the equational rewriter ultimately leads to many-page elaborate calculations of function definitions. While a number of folklore techniques are presented along the way, the treatment in this final part of the book provides students with few opportunities to acquire new skills.

Only from Chapter 10 on monads are on the table. Richard Bird first illustrates how elements of imperative programming (I/O, state, mutable arrays) lead to various monad instances. Chapter 11 then develops a library of monadic parsing combinators in the established Hutton-Meijer style. Both chapters feature particularly nice and practical examples (like an efficient variant of breadth-first search or the systematic conversion of context-free grammars into a composition of parsing combinators). Still, Bird remarks that "*[o]ur best advice is to use the monadic style sparingly... the most important aspect of functional programming, the ability to reason mathematically about its constructs, is lost.*"[3]

This piece of advice, along with many others, reminds us that this is an opinionated book. Bird has his list of "*Good Things to Use in Moderation*"—on which you will find monads just like `as`-patterns or operator sections—and he is not cautious to share it. I personally like this style and am convinced that these deliberate judgments, hints, and witty slogans ("*tupling is the dual of accumulating parameters*") serve the reader to gain a clear and memorable first picture of functional programming.

This is not a tutorial book on Haskell itself and it is not advertised as such. Throughout the entire book, Haskell is regarded as the vehicle, not the destination. From the very beginning, the reader is encouraged to experiment with Haskell and learn from the instant feedback given by the `ghci` REPL. Still, Haskell constructs that can be regarded core language features, like `newtype` or the use of (module-)qualified names, first appear in the late Chapters 11 and 12, respectively. Many abstractions that are pervasive in idiomatic Haskell code today (the `Applicative` type class or monad transformers, say) play no role at all. Richard Bird's emphasis is on fundamental programming techniques of wide applicability.

I do not want to close before I can underline what a treasure the book's collection of exercises presents. An extensive list of questions and programming assignments, always clearly connected to the discussion in the preceding chapter, closes each of the twelve chapters. Not a single exercise remains without a proposed answer or solution, rendering the book the ideal companion for self-study. These exercises are never dull and some are outright challenging. They have certainly inspired me as a teacher to rethink and improve the assignments I hand out.

[2] Through application of the rewriter to itself, remarkably.

[3] Again, the book provides perspective with a reference to "Just `do` it: Simple monadic equational reasoning", Jeremy Gibbons' and Ralf Hinze's ICFP 2011 paper.

Overall, I do not hesitate to highly recommend Richard Bird's new book on learning how to think functionally. It will certainly play a major role in my preparations for upcoming courses on functional programming. I am glad to see this series of textbook classics continue with another strong entry.

TORSTEN GRUST
Universität Tübingen