JFP **30**, e29, 3 pages, 2020.
doi:10.1017/S0956796820000258

Book Review

Review of "Algorithm Design with Haskell" by Richard Bird and Jeremy Gibbons, Cambridge University Press, 2020.
doi:10.1017/9781108869041

I strongly suspect that Richard Bird hides a magically productive book writing apparatus in his office. This time around, Bird pulled the machine's levers together with Jeremy Gibbons and out came *Algorithm Design with Haskell*, a book that is remarkable in many ways.

Algorithms in their purely functional form. Books on algorithms are plenty. Many present classical algorithms as entities carved in stone, conceived decades ago, and typeset in framed pseudo-code boxes. *Not so in the present book.* Bird and Gibbons treat any algorithm as a subject of calculation, to be derived systemically and elegantly starting from an (almost) blank sheet that initially holds nothing but a functional specification. In fact, several of these calculations start out with incantations like

$$solutions = filter\ good \cdot candidates \quad \text{or} \quad algorithm = aggregate \cdot test \cdot generate$$

which can then be transformed into an efficient implementation via fusion, or more generally, equational reasoning (except when the authors cannot—see below). The equational style also delivers the implementations' proofs of correctness which, otherwise, you rarely find in algorithm books.

Across the 430+ pages, I felt that deriving algorithms out of "thin air" (read: nothing but their specification) demystified them in the best possible sense: stepwise construction through composition of already known functions takes the place of genius algorithm engineers' *Eureka!* moments. In the capable hands of Bird and Gibbons, this feels empowering. Whole families of algorithms originate in one common functional specification (much like the two given above). While these are—quite amazingly, really—already executable, they are too inefficient for all practical purposes. Then, (1) cracking the composed functions open, (2) choosing their particular implementations, and (3) fusing the latter is what leads to a specific, efficient functional algorithm. The text goes as far as to identify "ritual steps" in algorithm calculation and, indeed, in the second half of the book recipes emerge that readers will be able to reuse when they tackle their own constructions.

Highlights. Bird and Gibbons have divided the book into six parts, five of which focus on particular algorithm design strategies: *divide and conquer*, *greedy algorithms*, *thinning*, *dynamic programming*, and *exhaustive search*. Interestingly, you will find the generic functional specification at the start of each part to serve as a guideline that tells whether your problem at hand will fit the upcoming discussion. From part to part, there is definitely

a build-up here and I think readers would be well advised to take in the text from front to back on a first encounter. Not least I would recommend to read the book cover to cover to make sure that you do not miss the bits of delightfully dry British humour that lurk throughout.

Generically, a greedy algorithm at each stage selects *the one* partial solution candidate with minimum cost and then proceeds to extend and complete it (or *greedy* = *minWith cost* · *candidates*, as the authors would say prior to fusion). Should multiple equally promising candidates exist, any deterministic *minWith* will pick one and thus bake the function's implementation decisions into algorithm calculation early on. "Too soon," argue the authors, and introduce *MinWith*, the non-deterministic uppercase sibling of *minWith*, to avoid such precommitment and instead select any of the minimum cost candidates. The text manages to preserve its style of reasoning in the face of this non-determinism—when required, "←" (or: *is one possible result of*) takes the place of "="—and stays away from going fully relational. Jeremy Gibbons goes on *YouTube*'s record to say that using a calculus of relations "is idealistically the right thing to do" but that would have led to a "pretty, but pretty complicated" result (in reference to *The Algebra of Programming*, Bird and de Moor, Prentice Hall, 1997). I call the present book's lighter weight approach to non-determinism a definite readability plus.

Non-deterministic functions then truly are in focus in the part on thinning, a true novelty in this book. Where greedy algorithms extend a single best candidate and exhaustive search considers all possible options, thinning strikes a middle ground and maintains a selection of promising partial candidates. Selection is embodied by a non-deterministic *ThinBy* abstraction which—only post-specification and -calculation—is refined into a concrete *thinBy* implementation late in the game. The authors recast a number of classical problems in terms of thinning, including ones like *knapsack* which otherwise have been tackled via dynamic programming for ages. These fresh—very confidently and competently presented—takes on established material are true highlights of the text.

We also find what now appears to be a staple of a Bird book: a fire hose of exercises, all of which come complete with solutions. Most of these exercises are derived directly from a chapter's ongoing exposition and thus naturally come with motivating context. Exercises touch on core, fun, and interesting parts of the discussion and *they can do so* since all answers are provided: no hole remains in the development should you fail to solve an exercise (and you likely will struggle, since Bird and Gibbons get you to work on everything from proofs of laws to algorithmic trickery). The sheer amount of quality content in the chapters' exercise postludes alone is worth the price of admission, if you ask me. The present book and Bird's earlier *Thinking Functionally with Haskell* (CUP, 2015) are certainly on par here.

This book and Haskell. From one-line specification to few-lines implementation, all algorithms in this book are expressed in Haskell and thus are immediately executable. Pseudo-code notation is nowhere to be found. The resulting conciseness is remarkable but also demanding for the reader: an essential fusion step can be presented in half a line but may take you half an hour to fully appreciate.

Still, the dialect of Haskell spoken by Richard Bird and Jeremy Gibbons is deliberately simple. Built-in types, "common-or-garden lists," and purely functional arrays suffice. The

https://doi.org/10.1017/S0956796820000258 Published online by Cambridge University Press

book's focus deliberately is on algorithms rather than the data structures they create and manipulate. Although there is great introductory material on symmetric and random access lists to be found in Part 1 of *Algorithm Design with Haskell*, Chris Okasaki's book on *Purely Functional Datastructures* (CUP, 2008) would make an ideal orthogonal companion. Bird and Gibbons are disciplined and lay out the operations they expect their data structures to support. Okasaki-supplied replacements of these data structures should thus readily drop in, I'd conjecture.

Haskell's `Applicative`, `Traversable`, or `Monad` go unused and the authors also abstain from the latter when it comes to the treatment of array updates or non-determinism. Just as in Bird's *Pearls of Functional Algorithm Design* (CUP, 2010), Haskell's laziness is ignored when the runtime of functions is discussed. What remains of Haskell, then, is its clarity and brevity. I was repeatedly reminded of simplehaskell.org and its effort to shine a light on the core of a language that keeps evolving rapidly. As a result, this is *not* a book on learning contemporary Haskell—and it never aims or claims to be. At the same time, *Algorithm Design with Haskell* is an understatement: any functional language could take the centre stage. Given this, *Thinking Functionally in Haskell* indeed leads up to and complements the present book. Should you study both, you will probably end up writing Haskell that has *Bird* stamped all over your code—there definitely are worse things to complain about, however.

There are few spots where you may experience a *déjà vu* if you have already read *Pearls of Functional Algorithm Design*. This certainly happened to me when the planning algorithm for the *Rush Hour* puzzle was developed. Often, however, what started out as a seemingly well-known story took an unexpected but welcome turn in *Algorithm Design with Haskell* (*maximum segment sum* viewed as a thinning problem tastes considerably different than its treatment in *Pearls*, for example). The intersection of both books is non-empty for sure, but where *Pearls* dives deeper into specific problems or puzzles, the new text represents a more ambitious and principled approach to algorithm design. I'd rather have both books on my shelf.

All in all: *replicate* 5 '∗'. I guess we may rest assured that both authors continue to keep the Oxford book machine well oiled. I certainly do look forward to what comes out next.

TORSTEN GRUST
Universität Tübingen
E-mail: torsten.grust@uni-tuebingen.de

https://doi.org/10.1017/S0956796820000258 Published online by Cambridge University Press