JFP **34**, e6, 4 pages, 2024.

doi:10.1017/S0956796824000042

Book Review

A review for the Journal of Functional Programming of Sandy Maguire, *Algebra-Driven Design* (Leanpub, 2020).

1 Context

Combinator libraries have been well studied in countless papers. Indeed, as Peyton Jones, Eber, and Seward wrote in 2000,

> At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library."
> (Peyton Jones et al., 2000)

So why a book now? Because those countless papers have been mostly accessible to functional programming experts, not the general software development community. Outside of functional programming, combinator libraries remain mostly unknown.

Maguire's book explains how to design combinator libraries using algebraic principles. The first half is about designing abstractions along with laws that characterize their semantics. The second half is about deriving implementations. The two halves work together to help fill the huge, mostly empty space on material about tackling complex, real-world problems with functional programming.

2 Overview

The book starts with a foreword that explains the value of rooting understandable abstractions in algebraic laws and lays out notational conventions for the Haskell code that follows.

The book's subsequent first half is about designing abstractions along with laws that characterize their semantics. It explain methodology using two examples—the first, comparatively simple one is a variant of Henderson's functional geometry (Henderson, 1982) called "tiles." The second one is a software system for supporting a "scavenger hunt" game.

With tiles, the book starts with the usual combinators on rotation and flipping, complementing their type signatures with algebraic laws. Here is the operator for clockwise rotation by 90°, followed by law stating that applying it four times gets back the original tile:

```
cw :: Tile -> Tile
∀ (t :: Tile).
  cw (cw (cw (cw t))) = t
```

The tiles part of the book gradually introduces more advanced combinators, culminating in an applicative functor. It carefully explains how to define equality for combinators in terms of observations. Different terms can produce identical-looking images of tiles as in the above law. Hence, equality needs to defined in terms of the images produced by—for example—rasterizing the image. If the rasterizations are the same for all possible resolutions, the tiles are equal.

The book's second example, the scavenger hunt, models a game played by (actual) people who run the city solving challenges (occasionally with the help of clues), logging proof of completed challenges with the system, and receiving reward points. For example, a player may be challenged to find "the three-fold monument [that] overlooks the city." The player (hopefully) finds the monument, proving this by providing geolocation data and a selfie, and receives an award. The game then progresses to the next challenge.

Scavenger hunt is a much more substantial example, and the design is not just presented but evolved over several revisions that tease out the combinator nature of the domain step by step. For example, a traditional model might come with a massive constructor for challenges like this:

```
pointOfInterest
    :: Clue
    -> Point
    -> Distance
    -> Reward
    -> Challenge
```

The book proceeds to separate out the individual aspects of this constructor into combinators, starting with adding a clue to a challenge:

```
clue :: Clue -> Challenge -> Challenge
```

More such constructors ensue, resulting in a substantial combinator model for challenges.

The second half of the book is about deriving implementations, revisiting the two examples of the first half. The tile implementation starts with an initial encoding ("deep embedding") and simplistic implementation and then swaps it out for a more realistic representation. Moreover, the book shows how to translate the laws to QuickCheck tests and write appropriate generators to make this effective. Thankfully, it also lays out how QuickSpec can automatically discover many of the laws constructed "manually" in the first half.

The book then proceeds to do the same for the scavenger hunt, eventually leading to an efficient, monad-based implementation of the game backend.

The book's third half is titled "Reference Material" and contains a tutorial on using QuickCheck and QuickSpec, as well as a summary of common algebraic laws.

The book's first half, on design, is careful to focus on the combinators and their laws in general terms. The brief introduction on Haskell notation from the beginning should be enough to make this part accessible, even to folks with little background in Haskell or even functional programming.

In the second half, more advanced Haskell features—specifically type classes—enter the scene: following this section and the material that follows really requires more Haskell knowledge, along with basic understanding of the concepts behind, say, the `Functor` und `Applicative` type classes.

3 Appraisal

Maguire’s book collects ideas that have been around the functional programming community for a long time, and it ties them together into a comprehensive methodology with compelling examples. The presentation is well organized and build concepts gradually.

The writing is casual yet precise and easy to understand. The code (a significant part of the book) also builds gradually from simple combinators to more advanced concepts, and it is available from a Github repository.

The nitty-gritty of QuickCheck generators, monad-based implementations, and SmallCheck is tremendously valuable, not only as resource for functional programmers but also as a treasure trove of teaching ideas and material.

Combinator libraries are applicable in many domains. Reactive animation, structured diagrams, stage lighting, and music are just a few examples from published papers. The work on financial contracts (Peyton Jones et al., 2000) has notably been the cornerstone of a commercial product. Many other industrial domains such as e-commerce shopping carts, factory production routes, and market data benefit from combinator models. Well-designed combinator models remain supple in the face of new requirements and greatly reduce the work needed to implement new features. Explaining how to construct them has tremendous value both for concrete software projects and for the field of software architecture in general.

Consequently, I wish the book would go further still and contrast algebra-driven design with traditional object-oriented design. For the scavenger hunt, Maguire tantalizingly hints at an actual, similar project, that produced “four rewrites” that never resulted in a “suitable design that simultaneously met all of the requirements and was easy to use.” Knowing what these rewrites were like, and why they failed, would strengthen the case the book is making. As a matter of course, combinator libraries would also emerge as radically different from object-oriented models, with unique advantages.

4 Conclusion

Maguire’s book is an important contribution that helps fill the huge, mostly empty space on tackling complex, real-world problems with functional programming. Its casual style and careful presentation make the first half suitable even for readers new to functional programming. Even if you are familiar with Haskell, combinator libraries and algebra, Maguire’s book will make you more effective in applying these ideas to real-world problems. Read it.

References

Henderson, P. (1982) Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP 1982, New York, NY, USA: Association for Computing Machinery, pp. 179–187. ISBN 0897910826. https://doi.org/10.1145/800068.802148.

Peyton Jones, S., Eber, J.-M. & Seward, J. (2000) Composing contracts: An adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, New York, NY, USA: Association for Computing Machinery, pp. 280–292. ISBN 1581132026. https://doi.org/10.1145/351240.351267.

MICHAEL SPERBER, *Active Group GmbH,*
Hechinger Straße 12/1, *72072 Tübingen, Germany*
(e-mail: sperber@deinprogramm.de)