Extracted from:

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

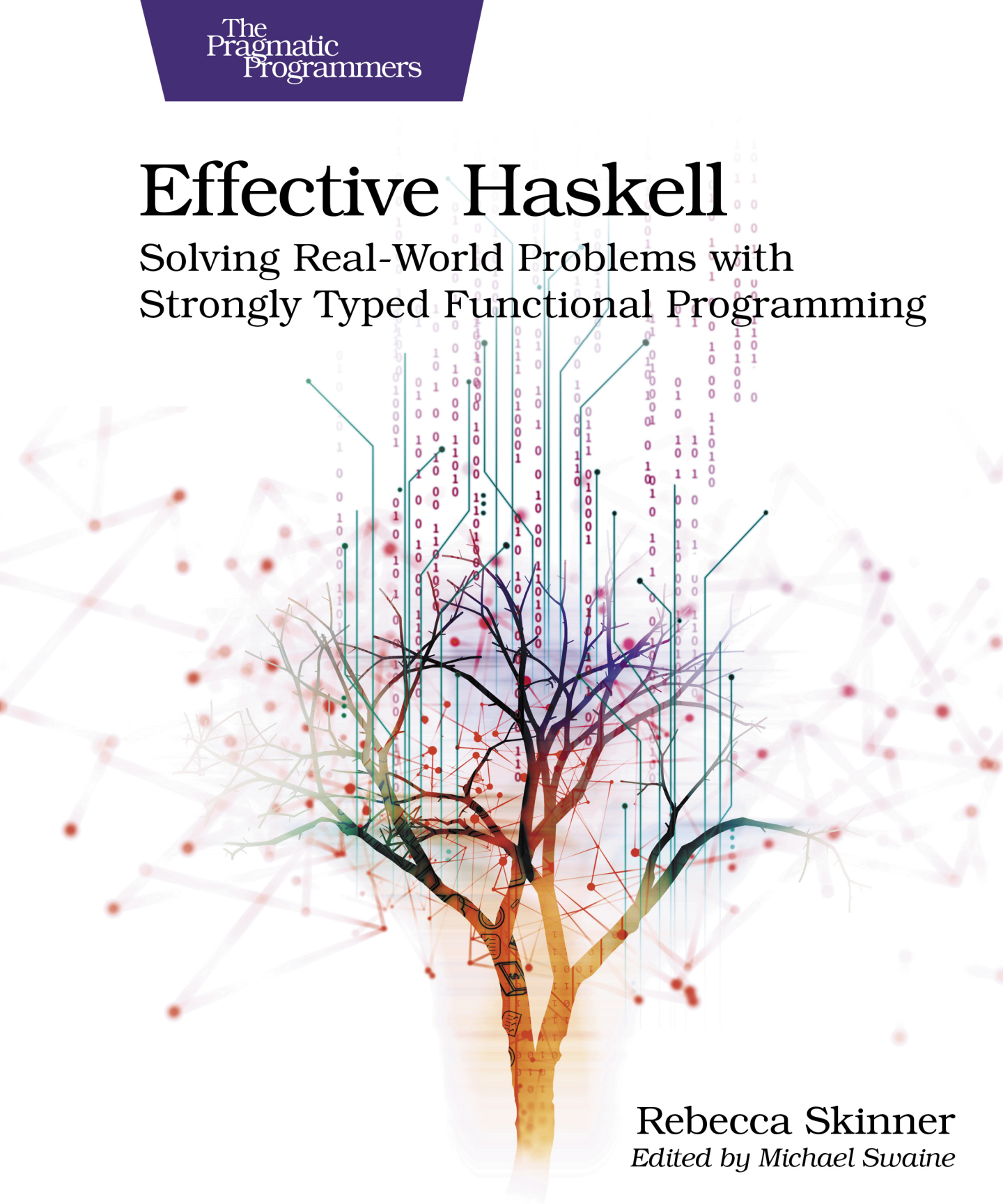
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming



Rebecca Skinner

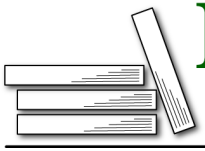
Edited by Michael Swaine

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

Rebecca Skinner

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-934-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2023

Introduction

Software development is harder than it's ever been, and the unfortunate reality is that every year things continue to get harder. Much of this difficulty is due to the complexity inherent in modern systems. Today, software needs to do more things, it needs to do them at a larger scale, and the consequences for failure are higher. To be effective in the market today, we have to use every tool at our disposal to rein in the complexity of our systems. I believe that Haskell is one of the greatest tools that we have at our disposal today to help us craft systems that are both more reliable and less complex.

Haskell isn't a new language. In fact, the first version of Haskell was published in 1990, a year before Python and five years before Java. In many ways Haskell has always been a remarkably successful language. It's been used widely in both industry and academia for the research and development of programming languages, and the design of Haskell has been incredibly influential in shaping other languages that are in wide use today.

Although it's been wildly successful as a research tool and programming language "influencer," industrial adoption of Haskell has lagged behind. Today, there are more Haskell jobs than ever, and more companies are choosing Haskell to build their products and key parts of their infrastructure. Right now, Haskell remains more of a secret weapon than a mainstream tool, but the clear benefits of Haskell for the kinds of systems that people are building today means that an inflection point in the popularity of the language is inevitable. If Haskell doesn't become the next big thing, then the next big thing will certainly end up looking even more like Haskell than any of the other myriad of languages that have been influenced by it.

Why Choose Haskell?

The reason that Haskell is such a good choice for modern software is that it gives us everything we need to build reliable, predictable, and maintainable systems that run efficiently and can be easily scaled horizontally. This is

thanks to Haskell's design as a lazy pure functional language with an exceptionally powerful and expressive static type system.

When you think of functional programming, the first thing that likely comes to mind are the sorts of functional programming features that are recently being added to mainstream object-oriented languages. These include things like:

- Support for closures or lambda functions
- Using functions like `map` and `reduce` instead of traditional loops
- Immutable data structures that copy their results rather than mutating their inputs

Haskell is different. The benefits we get from Haskell go far beyond being able to pass around functions and have immutable data structures. Thanks to its purity, Haskell can offer us strict guarantees about immutability across our entire program. This means that we never have to worry about unintended changes to shared or global state causing our program to crash, or think about how to coordinate access to mutable data.

One of the reasons that Haskell can give us strong guarantees about what our program does where other languages can't is thanks to the power of its type system. Haskell's type system is more expressive than the type system of any other mainstream programming language in use these days. Thanks to the type system, a Haskell program can keep track of information about what kind of data a variable holds, where it came from, what can be done with it, and even whether the function that calculates that value could possibly fail.

Of course, this type information doesn't just help us write better programs once. Every time we make changes to our program, the type checker does its job to ensure that we haven't introduced any new problems, and helps us track down things that might need to change. As applications grow and teams get larger, the power of the type system to help us refactor becomes even more important. Types become a way that we can communicate with our peers, to provide guard rails for how they use our code, and to make sure we are using the code they wrote as it was intended. In this way, Haskell doesn't just solve for the problems of complexity with the software we're writing, it also helps with the complexity inherent in building that software with a large team.

Why This Book

Haskell can offer enormous benefits to individuals and developers who want to write high quality software, but as the saying goes, "if it were easy, everyone would do it." The benefits of Haskell come at the cost of a steep learning curve. Haskell is hard to learn, but this book will help.

Learning Haskell can be hard in part because it's so different from other languages you've probably used. This ends up being a particularly hard problem because many of the most unusual concepts that you need to learn often show up all at the same time, leading to circular dependencies in your learning plan. This book has been carefully designed, especially in the first half, to provide an on-ramp to the language that avoids the need to get into circular knowledge references.

Rather than teach you how to translate your programs from other languages, in this book you'll develop an intuition for how to think about programs in Haskell from the ground up. This will make it easier for you to read other developers' code, make you more effective at writing code, and help you with troubleshooting.

Most of the chapters in this book build on concepts from previous chapters, and no content in any chapter relies on concepts that have not yet been introduced. You'll never be forced to use something that hasn't yet been fully explained. In the last half of this book, once you have worked through the fundamental materials, you may be able to approach some material out of order if there are particular areas that you are interested in.

Some features of Haskell can seem unnecessarily complex the first time you encounter them. Some people, when they are faced with a feature that makes no sense, will assume the feature was a bad idea and give up on learning it altogether. Other people will put the concept on a pedestal and assign it disproportionate significance. In either case, the lack of motivation for the things Haskell does differently can be a barrier to learning. To help with that, each time a new concept is introduced in this book, we'll dedicate a significant amount of time to establishing a motivation for that concept to help you better internalize the reason for the design decisions Haskell makes. Understanding the motivation will ensure you're better positioned to make informed choices about how to design your applications, and when and how to use features of the language.

Since you'll be learning to think about programming in an entirely new way, we'll approach the material quite slowly in the beginning, carefully outlining all of the intermediate steps that go into executing some code and walking through multiple examples. As you approach the middle of the book, the pace will pick up, and by the last few chapters you should be learning new concepts at the pace of a native Haskell developer.

What to Expect as You Read

This book focuses on teaching through the demos and hands-on example code. Most of the chapters in this book will start with a motivating example followed

by several interactive demonstrations of a concept that you can reproduce using the interactive Haskell development environment `ghci`. Most chapters will also include some projects you can build as you are working through material. The chapters will include all of the code you need to build a functioning minimal example, but you are encouraged to make modifications and experiment with the code as you are working through the book. At the end of each chapter you'll also have some exercises that build on the examples you wrote. These examples will help you learn how to navigate Haskell's documentation and work within its ecosystem to self teach, so you are better equipped to continue learning after you've finished the book.

Compared to other programming language communities, parts of the Haskell community can tend to be a little "math jargon" heavy. It's not uncommon to see terms from theoretical computer science and math make their way into blog posts, articles, and even library documentation. This book aims to teach Haskell without requiring either a strong background in mathematics or familiarity with mathematical jargon. Since knowing the jargon and getting comfortable using it will ultimately help make you a more effective Haskell developer, common jargon terms will be introduced, defined, and then used consistently throughout the book. If you are skipping ahead and see some intimidating sounding language, turn back a few pages and you're likely to find a definition and several examples to help you make sense of the words before they start being used regularly.

How to Read This Book

This book has been designed to be read cover-to-cover as a tutorial and workbook, or to be used in a classroom or reading group setting. Starting with Chapter 1, each new chapter will continue a theme or build on some knowledge that you picked up in the previous chapter. If you have some prior experience with Haskell, it's worthwhile to start reading from the beginning so that you can follow along with the subsequent references to earlier material.

For more experienced Haskell developers, this book can also serve as a useful resource to help you learn some practical ways to apply more advanced techniques. If you've used Haskell in school or written a few small programs and are looking to move into building larger production applications, you may find it helpful to skim the first half of this book and then start reading the second half more thoroughly.

As you are working through the book, you'll encounter several different kinds of example code. You should always be able to tell what type of environment you should be working in based on the formatting of the examples:

- Code that starts with a λ character should be typed into ghci.
- Lines that start with user@host$ should be typed into a shell like bash or zsh.
- Other code can be written in Haskell files using your text editor, or written directly into ghci at your discretion.

Until you have finished Chapter 5, create a new directory for each chapter. Inside of the directory you create for each chapter, create a file named after the current chapter, for example, Chapter1.hs. You can use this for keeping track of example code and experiments you want to run. You'll also create several files named Main.hs as you are working through the examples. You can put each of these in a subdirectory, for example, one subdirectory per chapter, or you can rename your old Main.hs files when you are no longer actively working through them. Whatever organizational scheme you prefer, ensure you keep around all your examples and experiments since you'll want to refer to them frequently as you are learning.

Once you've worked through the chapter on Cabal on page ? you'll be better equipped to create fully stand-alone projects that you can build. You'll also learn how to re-use code that you've written. From that point onward, you can create a new project for each chapter or each major example.

Following Along with Example Code

As you read this book, you'll work through examples iteratively, making changes to earlier code and adding new features. Once you've learned about how to import code from other modules, we'll begin introducing new features iteratively that require adding additional imports. Similarly, once you've learned how to work with language extensions, we'll add them as we work through examples. The rest of this section will discuss how we'll approach introducing new imports and extensions in example code. Don't worry too much about the syntax yet. As you work through the book, you'll learn about imports and language extensions before they are required for any examples. For now, skim this section and feel free to come back to it later if you need to.

Most of the chapters in this book will focus on building up a few small example programs. In some cases, we'll explicitly define a new module when we're starting a new example. In this case, these new modules may start out including a few language extensions or imports.

```
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE DerivingStrategies #-}
module Main Where

import Data.Text (Text)

main :: IO ()
main = print helloWorld
```

As we iterate through the example, we'll add new features that might require additional language extensions or add-ons. When we're getting ready to use a new module or extension, we'll add them to the top of an example:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.ByteString (ByteString)

helloWorld :: ByteString
helloWorld = "Hello, World"
```

In your own code, you should add these to the relevant parts of your module. Here's an example of what your own code should look like as you follow along with the examples:

```
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE OverloadedStrings #-}
module Main Where

import Data.ByteString (ByteString)
import Data.Text (Text)

helloWorld :: ByteString
helloWorld = "Hello, World"

main :: IO ()
main = print helloWorld
```

Not all of the examples that you work through will start with a module and a set of imports or extensions. In these cases, you can start with your own empty module, or you can work though the examples in `ghci`.

Compiler Versions, Language Standards, and Extensions

Although there have been several different implementations of Haskell over the years, the Glasgow Haskell Compiler (GHC) is the de facto standard Haskell compiler. In this book we'll focus on Haskell as implemented by GHC 9.4, which is the newest stable release at the time of this writing. All of the examples have also been tested with GHC 8.10.

Compiler Version Differences

A few examples in this book will use newer features of GHC not available in version 8.10. Look out for an aside, like this one, to learn about newer features and how to write code without those features when you need to support older compilers.

As Haskell evolves, new features are typically added through *extensions*. Language extensions allow you to enable and disable specific language features. The Haskell2010 language standard is the default language version

that's used by GHC 8.10, and it includes a number of extensions that are enabled by default. In GHC 9.4, the GHC2021 language version is used by default. GHC2021 isn't an officially published Haskell standard; instead it represents a number of commonly accepted GHC specific nonstandard extensions to Haskell2010 that are enabled by default.

In this book, we'll target Haskell2010. Any language extensions that aren't included in Haskell2010 will be introduced and discussed. Complete example programs will always include all extensions that would be required when using Haskell2010. Shorter examples may omit language extensions for the sake of readability.

GHC2021 Extensions

We'll target Haskell2010 as a baseline when choosing which language extensions to highlight in this book. If you're using GHC2021, look for an aside like this to tell you when an extension is included by default and doesn't need to be enabled explicitly.

Extension	Enabled
AllowAmbiguousTypes	Manually
BangPatterns	GHC2021
ConstraintKinds	GHC2021
DataKinds	Manually
DefaultSignatures	Manually
DeriveAnyClass	Manually
DerivingStrategies	Manually
DerivingVia	Manually
ExistentialQuantification	GHC2021
ExplicitForAll	GHC2021
FlexibleContexts	GHC2021
FlexibleInstances	GHC2021
FunctionalDependencies	Manually
GADTs	Manually
GeneralizedNewtypeDeriving	GHC2021
KindSignatures	GHC2021
MultiParamTypeClasses	GHC2021
OverloadedStrings	Manually
PolyKinds	GHC2021

Extension	Enabled
QuantifiedConstraints	Manually
RankNTypes	GHC2021
RecordWildCards	Manually
ScopedTypeVariables	GHC2021
TupleSections	GHC2021
TypeApplications	GHC2021
TypeFamilies	Manually
TypeOperators	GHC2021
UndecidableInstances	Manually
NoStarIsType	Manually
PolyKinds	GHC2021
StandaloneDeriving	GHC2021

Libraries and Library Versions

The examples in this book stick to the standard library, `base`, as much as possible. For features that aren't available in `base`, we'll stick to a small selection of popular libraries. This should ensure maximum compatibility, at the cost of not showing off some very interesting libraries that are worth learning. The following table includes the exact versions of each package that were used for the examples. In any cases where there are incompatible changes between library versions, we'll use the most recent version of the library.

Package	Version (GHC 9.4)	Version (GHC 8.10)
base	4.17.0.0	4.14.3.0
bytestring	0.11.3.1	0.10.12.0
base64-bytestring	1.2.1.0	1.2.1.0
text	2.0.1	1.2.4.1
containers	0.6.6	0.6.5.1
vector	0.12.3.1	0.12.3.1
time	1.12.2	1.9.3
unix	2.7.3	2.7.2.2
mtl	2.2.2	2.2.2
transformers	0.5.6.2	0.5.6.2
process	1.6.15.0	1.6.13.2