The Joy of Haskell

Finding Success and Failure

Julie Moronuki

Chris Martin

Finding Success (and Failure) in Haskell
by Julie Moronuki and Chris Martin

2018-12-02: First draft
2019-04-02: Second draft
2019-05-21: First edition

Contents

Preface **7**

1 Introduction to case expressions **11**

1.1 Conditionals 12

1.2 Reading type signatures 12

1.3 Branching patterns 16

1.4 Case expressions 16

1.5 Sum types 18

1.6 Exercises 19

2 Case expressions practice **23**

2.1 The anagram checker 24

2.2 The word validator 26

2.3 Validate first, then compare 29

2.4 Interactive program 30

2.5 Exercises 33

3 Validation functions **37**

3.1 Project setup 38

3.2 checkPasswordLength 40

3.3 requireAlphaNum 42

3.4 cleanWhitespace 44

3.5 Exercises 47

4 The Maybe Monad 50

4.1 Combining the validation functions 50

4.2 De-nesting with infix operators 54

4.3 Enter the monad 56

4.4 Classes and instances 58

4.5 TypeApplications 58

4.6 Cases and binds 60

4.7 Exercises 61

5 Refactoring with Either 65

5.1 Adding error messages 65

5.2 Introducing Either 67

5.3 The Either Monad 68

5.4 Using Either 71

5.5 Exercises 73

6 Working with newtypes 78

6.1 Introducing newtypes 78

6.2 Declaring new types 81

6.3 Using our new types 82

6.4 Revising main 85

6.5 Exercises 87

6.6 Notes on monadic style 90

7 Introducing Applicative 92

7.1 Validating usernames 92

7.2 Adding to main 93

7.3 Constructing a User 94

7.4 Constructors are functions 97

7.5 Using Applicative 99

7.6 Exercises 100

8 Refactoring with Validation 104
- 8.1 Introducing validation 104
- 8.2 Adding a dependency 107
- 8.3 Nominal refactoring 108
- 8.4 Interpreting the errors 110
- 8.5 An Error semigroup 111
- 8.6 Using Applicative 114
- 8.7 Exercises 117

9 Better error messages 122
- 9.1 The problem 122
- 9.2 The error functions 124
- 9.3 Gathering up the errors 125
- 9.4 Lists upon lists 126
- 9.5 Coercion 128
- 9.6 Handling success 129
- 9.7 The final main 131
- 9.8 Exercises 132

10 Coercible 136
- 10.1 Enter Coercible 138
- 10.2 What can be coerced? 138
- 10.3 Updating the display function 140
- 10.4 Type applications 141
- 10.5 Coercibility is transitive 142
- 10.6 Coercion via type parameters 144
- 10.7 Coercing functions 144

11 Generalizing further 148
- 11.1 Designing a typeclass 149
- 11.2 Folding over sum types 151
- 11.3 Desire for a generalized fmap 155
- 11.4 The lens library 157

11.5 The Success and Failure prisms 158
11.6 The Either-Validation isomorphism 162
11.7 The Validate class 164
11.8 Exercises 168

A Solutions to exercises 171

B API reference 192

C GHCi command reference 200

Preface

Julie originally planned this course and taught a version of it at the Austin Haskell Meetup. The group had, by then, learned about monads and applicatives and how typeclasses work and all that good stuff, but it wasn't yet clear and concrete to them. It's one thing to talk about an idea and another to make use of it, so this series of lessons was planned to understand some things about monads and applicatives ***by using them***. We started with a basic problem to solve – validating some user inputs – wrote a few basic functions and, over the course of a few hours, refactored it to use different types. Some of those types are monads, and some are not, and we were able to reach a concrete understanding of why and why not.

We've since revised and refined the course and edited the code to illustrate several additional core Haskell concepts while still being able to introduce them one at a time, to keep things tractable. By starting with basic language concepts (`if-then-else` and `case`) and growing a single example gradually, we made a book that is accessible to beginners, practical, and helpful to anyone who wants to get started writing programs in Haskell.

This book is for people who have just started getting into Haskell but would like to move quickly and understand by doing. We assume very little prior knowledge of Haskell. We work through examples without understanding theory or how and why things work too deeply. We give ***just***

enough information, just at the time when you need it.

Success and Failure Most programming languages have, in some form or other, a way of dealing with failure – or more specifically, a way to combine multiple smaller programs that might fail into a larger program that might fail. In an imperative style, this happens by executing the program's instructions in sequence and halting when an error occurs. Since the instruction failed to produce its value or effect, which was presumably necessary for the rest of the program, execution can continue no further, and whatever error information was produced by the failed subprogram constitutes the result of the program overall.

The deficiency of the process described above is that it doesn't always provide us with as much information as we might like when failure occurs. Because execution halts immediately, this approach can only ever give us information about the ***first*** problem that was encountered. Careless application of this error handling mechanism can give rise to unfortunate software behavior. Consider situations in which a user must fill out a form that will be checked programmatically for mistakes. An ideal program might show the user a list of ***every*** problem on the form; a flawed program may show only the first.

This is the problem that motivates this book. It is one that many programmers have encountered and, to our knowledge, one to which only functional programming with typeclasses permits a straightforward general solution in which writing the ideal program is no more difficult than writing the flawed one.

Programmers ask ***Why monads***? This is why: the monad class is our tool for generalizing the notion of "program" beyond "run a series of subprograms until one fails." There is a whole world of other kinds of programs; in this book, we are concerned with programs for which the execution model is "run ***all*** of the subprograms, and if any of them fail, produce a list of all of the failures." Once we understand how the `Monad` and `Applicative` classes generalize programs, the solution to our problem falls almost effortlessly into our lap.

The book begins with two chapters on `case` expressions to ensure a solid foundation. From there, we write three functions for checking that inputs are valid passwords according to the rules of our system. The rest of the book iteratively expands on and refactors those functions into a small program that validates usernames and passwords, constructs a `User` (the product of a username and a password) if both are valid inputs, and returns pretty error messages if they are not. Along the way we learn about `Monad` and `Applicative`, how they are similar, how they differ, and how to use types to rethink our solutions to problems.

Follow along We encourage you to follow along with the steps that we take in this book, type all of the code yourself, and do the exercises at the end of each chapter.

You will learn to build a Haskell project with an executable. The only thing you'll need to install is Stack; learn about Stack and how to install it at `https://haskellstack.org`. Stack will take care of installing the Haskell compiler, GHC, automatically. If you're already comfortable building a project by other means, such as with `cabal-install` or Nix, then you can still follow along, although we'll assume that you are able to adapt the instructions for your build system of choice.

GHC comes with a REPL ("read-evaluate-print loop") called GHCi ("GHC interactive") which makes it easy to run quick experiments to try things out.

Each chapter except one ends with exercises. Some are fairly straightforward extensions of what we've just done in the chapter, while others introduce new concepts. In general, they are ordered by difficulty, with the first exercises in the chapter being the most familiar and the last one most likely being the most challenging, probably introducing a new concept or giving you the least amount of help. A few stretch way beyond the current text to introduce entirely new libraries to encourage you to get closer to idiomatic Haskell. You should be able to adequately follow the main body of the text, however, without doing those exercises, so do not feel obligated to complete them all before moving on to the next chapter.

Chapter 1

Introduction to case expressions

This chapter compares two kinds of expressions in Haskell:

1. Conditional `if-then-else` expressions, which you are probably familiar with from other programming languages; and

2. `case` expressions, which serve a similar ***branching*** role as conditionals, but with much more generality.

Conditionals and `case` expressions serve a similar purpose: they both allow a function's behavior to vary depending on the value of an expression. However, `case` expressions have some flexibility that `if` expressions do not have, namely allowing behavior to branch on values other than booleans.

Index

`*`
the kind of types, 76
`++`
list concatenation, 112
`.`
function composition, 55
`:`
list constructor, 45
`:|`
non-empty list constructor, 134
`<*>`
applicative operator, 96
`<*`
left bird, 115
`*>`
right bird, 115, 116, 118
`<-`
in `do` blocks, 31, 86
`<>`
semigroup operator, 54, 111
`=>`
typeclass constraint, 14, 106
`>>`
monadic sequencing, 89
`>>=`
monadic bind, 56, 90
`|`
in data declarations, 19, 67
`_`
hole, 24
type wildcard, 69

absolute value, 19
all, 27
anagram, 23
Applicative, 96, 101
IO instance, 102
Validation instance, 106
do, 119
arity, 76

associativity, 55

Bifunctor, 156
Bool, 17
  folding over, 152

.cabal file, 38, 107
case expression, 18
catamorphism, 153
checkAnagram, 30
checkPasswordLength, 41, 71, 82
checkUsernameLength, 83
class declaration
  Applicative, 101
  FoldAB, 154
  LiftAB, 149
  MaybeAB, 150
  Semigroup, 111
  Validate, 164
cleanWhitespace, 44, 83
coercion, 138
  contrast with isomorphism, 163
Control.Lens, 157
“Couldn’t match ...”, 51, 129

data constructor, 17
Data.Bifunctor, 156
Data.Bool, 152
Data.Char, 27
Data.Coerce, 138
Data.List, 25
Data.List.NonEmpty, 134
Data.Semigroup, 111
Data.Text, 119
Data.Validation, 105
dependency, 107
display, 130, 140, 141, 151, 155
do, 31, 62, 90, 118

Either, 67
  monadic composition of, 68
  Monad instance, 69
  as validation output, 71
  folding over, 154
Error, 82, 108, 113, 133, 135
  errorCoerce, 128
  Semigroup instance, 111

Failure
  data constructor, 105
  prism, 157, 159
-fdefer-typed-holes, 24
first, 156
fmap, 87, 156, 159
FoldAB, 154
folding
  over list, 27
  over `Bool`, 152
  over `Maybe`, 153
  over `Either`, 154
  over `Validation`, 154

getLine, 31

GHCi, 12

head, 21
hole, 24

if … then … else, 12, 151
:info GHCi command, 17, 82
instance declaration, 58, 60
    Monad Maybe, 69
    Monad Either, 69
    Applicative Validation, 106
    Semigroup Error, 111
    Semigroup [ ], 112
    Validate Either, 165
    Validate Validation, 166
IO
    monadic composition of, 61
    applicative composition of, 102
isAlpha, 28
isAlphaNum, 42
isAnagram, 26
Iso, 162
    simple, 162
    construction of, 165
isomorphism, 104, 162
isSpace, 45
isWord, 27, 28

:kind GHCi command, 76

LANGUAGE pragma, 59, 120
length, 41
lens, 157
let, 86
LiftAB, 149
:load GHCi command, 13

main
    module, 13, 39
    IO action, 30, 39
makeUser, 99, 125, 164, 166
map, 35
mapFailure, 155
Maybe, 18
    as validation output, 41
    monadic composition of, 56
    Monad instance, 69
    folding over, 153
MaybeAB, 150
Monad, 57
    Either instance, 69
    IO instance, 61
    Maybe instance, 69
monomorphism, 15

\n, 32
newtype, 80, 138
    deriving, 113
"No instance for …", 58, 110
NonEmpty, 134
null, 21, 27
Num, 14

optics, 158
OPTIONS_GHC pragma, 24
Ord, 14
over, 158, 162

palindrome, 33
parentheses, 20
partial function, 17, 21
passwordErrors, 124, 156, 160
polymorphism, 15, 73, 112, 141, 167
preview, 161
Prism, 158
    simple, 161
product type, 94
pure, 101
putStr, 32

:reload GHCi command, 15
representational equality, 81
requireAlphaNum, 42, 43, 83, 109, 145
reverse, 33
review, 163, 164
Rule, 146

safeHead, 22
safeTail, 22
second, 156
Semigroup, 106, 111
:set GHCi command, 24, 59
sort, 25
stack new, 38
stack repl, 12, 39
    –package, 105
String, 25
    gaps, 66
    overloaded, 120
Success
    data constructor, 105
    prism, 157, 158
sum type, 19, 67

tail, 21
testing, 74
Text, 119
tuple, 20, 95
type ambiguity, 146, 168
type application, 59, 141, 167
type constructor, 19
:type GHCi command, 13, 82
type synonym, 79, 146
typeclass, 14, 58, 149

undefined, 24
unlines, 127
usernameErrors, 125

Validate, 164
validatePassword, 52, 56, 65, 72, 85, 116, 147
validateUsername, 92
Validation, 105
    lack of Monad instance, 106
    Applicative instance, 106

as validation output, 109
folding over, 154
Iso, 164
view, 163