contents

foreword xiii
preface xv
acknowledgments xvi
about this book xvii

PART 1 INTRODUCTION TO FUNCTIONAL PROGRAMMING 1

1 *What is functional programming? 3*

1.1 The benefits of FP: a simple example 4

A program with side effects 4 ▪ A functional solution: removing the side effects 6

1.2 Exactly what is a (pure) function? 9

1.3 Referential transparency, purity, and the substitution model 10

1.4 Summary 13

2 *Getting started with functional programming in Scala 14*

2.1 Introducing Scala the language: an example 15

2.2 Running our program 17

2.3 Modules, objects, and namespaces 18

2.4 Higher-order functions: passing functions to functions 19

A short detour: writing loops functionally 20 ▪ Writing our first higher-order function 21

2.5 Polymorphic functions: abstracting over types 22

An example of a polymorphic function 23 ▪ Calling HOFs with anonymous functions 24

2.6 Following types to implementations 25

2.7 Summary 28

3 *Functional data structures 29*

3.1 Defining functional data structures 29

3.2 Pattern matching 32

3.3 Data sharing in functional data structures 35

The efficiency of data sharing 36 ▪ Improving type inference for higher-order functions 37

3.4 Recursion over lists and generalizing to higher-order functions 38

More functions for working with lists 41 ▪ Loss of efficiency when assembling list functions from simpler components 44

3.5 Trees 44

3.6 Summary 47

4 *Handling errors without exceptions 48*

4.1 The good and bad aspects of exceptions 48

4.2 Possible alternatives to exceptions 50

4.3 The Option data type 52

Usage patterns for Option 53 ▪ Option composition, lifting, and wrapping exception-oriented APIs 56

4.4 The Either data type 60

4.5 Summary 63

5 *Strictness and laziness 64*

5.1 Strict and non-strict functions 65

5.2 An extended example: lazy lists 68

Memoizing streams and avoiding recomputation 69 ▪ Helper functions for inspecting streams 69

5.3 Separating program description from evaluation 70

5.4 Infinite streams and corecursion 73

5.5 Summary 77

6 *Purely functional state* *78*

6.1 Generating random numbers using side effects 78
6.2 Purely functional random number generation 80
6.3 Making stateful APIs pure 81
6.4 A better API for state actions 84
Combining state actions 85 ▪ Nesting state actions 86
6.5 A general state action data type 87
6.6 Purely functional imperative programming 88
6.7 Summary 91

PART 2 FUNCTIONAL DESIGN AND COMBINATOR LIBRARIES...93

7 *Purely functional parallelism* *95*

7.1 Choosing data types and functions 96
A data type for parallel computations 97 ▪ Combining parallel computations 100 ▪ Explicit forking 102
7.2 Picking a representation 104
7.3 Refining the API 105
7.4 The algebra of an API 110
The law of mapping 110 ▪ The law of forking 112 Breaking the law: a subtle bug 113 ▪ A fully non-blocking Par implementation using actors 115
7.5 Refining combinators to their most general form 120
7.6 Summary 123

8 *Property-based testing* *124*

8.1 A brief tour of property-based testing 124
8.2 Choosing data types and functions 127
Initial snippets of an API 127 ▪ The meaning and API of properties 128 ▪ The meaning and API of generators 130 Generators that depend on generated values 131 ▪ Refining the Prop data type 132
8.3 Test case minimization 134
8.4 Using the library and improving its usability 136
Some simple examples 137 ▪ Writing a test suite for parallel computations 138
8.5 Testing higher-order functions and future directions 142

8.6 The laws of generators 144
8.7 Summary 144

9 *Parser combinators 146*

9.1 Designing an algebra, first 147
9.2 A possible algebra 152
Slicing and nonempty repetition 154
9.3 Handling context sensitivity 156
9.4 Writing a JSON parser 158
The JSON format 158 ▪ A JSON parser 159
9.5 Error reporting 160
A possible design 161 ▪ Error nesting 162
Controlling branching and backtracking 163
9.6 Implementing the algebra 165
One possible implementation 166 ▪ Sequencing parsers 166
Labeling parsers 167 ▪ Failover and backtracking 168
Context-sensitive parsing 169
9.7 Summary 171

PART 3 COMMON STRUCTURES IN FUNCTIONAL DESIGN......173

10 *Monoids 175*

10.1 What is a monoid? 175
10.2 Folding lists with monoids 178
10.3 Associativity and parallelism 179
10.4 Example: Parallel parsing 181
10.5 Foldable data structures 183
10.6 Composing monoids 184
Assembling more complex monoids 185 ▪ Using composed monoids to fuse traversals 186
10.7 Summary 186

11 *Monads 187*

11.1 Functors: generalizing the map function 187
Functor laws 189

11.2 Monads: generalizing the flatMap and unit functions 190

The Monad trait 191

11.3 Monadic combinators 193

11.4 Monad laws 194

The associative law 194 ▪ Proving the associative law for a specific monad 196 ▪ The identity laws 197

11.5 Just what is a monad? 198

The identity monad 199 ▪ The State monad and partial type application 200

11.6 Summary 204

12 ***Applicative and traversable functors 205***

12.1 Generalizing monads 205

12.2 The Applicative trait 206

12.3 The difference between monads and applicative functors 208

The Option applicative versus the Option monad 209
The Parser applicative versus the Parser monad 210

12.4 The advantages of applicative functors 211

Not all applicative functors are monads 211

12.5 The applicative laws 214

Left and right identity 214 ▪ Associativity 215
Naturality of product 216

12.6 Traversable functors 218

12.7 Uses of Traverse 219

From monoids to applicative functors 220 ▪ Traversals with State 221 ▪ Combining traversable structures 223 ▪ Traversal fusion 224 ▪ Nested traversals 224 ▪ Monad composition 225

12.8 Summary 226

PART 4 EFFECTS AND I/O 227

13 ***External effects and I/O 229***

13.1 Factoring effects 229

13.2 A simple IO type 231

Handling input effects 232 ▪ Benefits and drawbacks of the simple IO type 235

13.3 Avoiding the StackOverflowError 237

Reifying control flow as data constructors 237
Trampolining: a general solution to stack overflow 239

13.4 A more nuanced IO type 241

Reasonably priced monads 242 ▪ A monad that supports only console I/O 243 ▪ Pure interpreters 246

13.5 Non-blocking and asynchronous I/O 247

13.6 A general-purpose IO type 250

The main program at the end of the universe 250

13.7 Why the IO type is insufficient for streaming I/O 251

13.8 Summary 253

14 ***Local effects and mutable state 254***

14.1 Purely functional mutable state 254

14.2 A data type to enforce scoping of side effects 256

A little language for scoped mutation 256 ▪ An algebra of mutable references 258 ▪ Running mutable state actions 259
Mutable arrays 262 ▪ A purely functional in-place quicksort 263

14.3 Purity is contextual 264

What counts as a side effect? 266

14.4 Summary 267

15 ***Stream processing and incremental I/O 268***

15.1 Problems with imperative I/O: an example 268

15.2 Simple stream transducers 271

Creating processes 272 ▪ Composing and appending processes 275 ▪ Processing files 278

15.3 An extensible process type 278

Sources 281 ▪ Ensuring resource safety 283 ▪ Single-input processes 285 ▪ Multiple input streams 287 ▪ Sinks 290
Effectful channels 291 ▪ Dynamic resource allocation 291

15.4 Applications 292

15.5 Summary 293

index 295